IoT Requirements and Architecture for Personal Health

Wyatt Lindquist

This dissertation is submitted for the degree of

Doctor of Philosophy

School of Computing and Communications

Lancaster University, UK

December 2020

DECLARATION

I declare that the work presented in this thesis is, to the best of my knowledge and belief, original and my own work. The material has not been submitted, either in whole or in part, for a degree at this, or any other university. Any user research in this thesis was conducted ethically and with consent.

Wyatt Lindquist

To my Mom & Dad.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Professor Sumi Helal, for his endless support, guidance, and advice throughout my PhD research. I am truly grateful to have met Prof. Helal in the University of Florida Operating Systems course; he opened many doors for me, introducing me to the IoT research area and giving me the amazing opportunity to complete my PhD degree at Lancaster University. Prof. Helal's knowledge and intuition regarding Health IoT and academics in general is truly an inspiration, and will undoubtedly have a lasting impact on me.

I would also like to thank Dr. Ahmed Khaled for working closely with me throughout my PhD and introducing me to his own research in personal IoT, as well as all of my colleagues in the School of Computing and Communication; I am grateful to Dr. Gerald Kotonya, Dr. Jaejoon Lee, Dr. Chris Bull, Dr. Wenjie Ruan, Dr. Jiangtao Wang, Dr. Mahsa Honary, Dr. Emma Wilson, and Wesley Hutchinson for their insight and collaboration through the Lancaster Digital Health Group. I also would like to thank my coworkers in Room C21, Dr. Roberto Rodrigues Filho, Dr. Abdessalam Elhabbash, Dr. Vatsala Nundloll, Dr. Richard Basett, Dr. Irni Khairuddin, Paul Dean, and especially Alex Wild, whose friendship made my time at Lancaster infinitely more interesting and enjoyable.

Finally, I would like to thank my family for their endless support throughout my work, as well as my close friends in the United States for staying in touch despite the difference in time and availability. In special, I would like to thank Nick Cioli for working with me on all sorts of side projects and for keeping me sane during the pandemic, Kevin Warren for always playing games or watching shows with me, and Ian Calder for coming a great distance to visit me.

Thank you all,

Wyatt Lindquist

IoT Requirements and Architecture for Personal Health
Wyatt Lindquist

Submitted for the degree of *Doctor of Philosophy*
December 2020

ABSTRACT

Personal health devices and wearables have the potential to drastically change the current landscape of wellness and care delivery. As these devices become commonplace, more and more patients are gaining access to new forms of simplified health monitoring and data collection, empowering them to engage in their own health and well-being in unprecedented ways. Cheap and easy-to-use *health IoT* devices are leading the transformation towards a continuum-of-care health system—focused on detection and prevention—where health issues can be caught before hospital care or professional intervention is needed. However, this vision is set to outpace the expectations and capabilities of today's connected health devices, challenging existing ecosystems with unique requirements on functionality, connectivity, and usability.

This thesis presents a set of health IoT requirements that are especially relevant to the design of a connected device's low-level software features: its *thing* architecture. These requirements represent shared concerns in health-related IoT scenarios that can be solved with the features and capabilities of smart *things*. The thesis presents an architectural design and implementation of concrete features influenced by some of these requirements—leading to the Atlas Health IoT Architecture—which explores the role of safe and meaningful interactions between devices and users, referred to as *IoTility*. The thesis also considers the IoTility of smartphone applications in health scenarios, called Mobile Apps As Things (MAAT), resulting in a programming enabler that more closely integrates app features with those of physical *thing* devices. Alongside these implementations, this thesis presents a set of experimental evaluations investigating the feasibility of both MAAT and the architectural requirements as a whole.

PUBLICATIONS

- **Wyatt Lindquist**, Sumi Helal, Ahmed Khaled, Gerald Kotonya, and Jaejoon Lee. 2019. MAAT: Mobile Apps As Things in the IoT. ACM Journal on Interactive, Mobile, Wearable, and Ubiquitous Technologies, vol. 3 no. 4, article 143. December 2019. Also presented as a full paper in the main track of the ACM International Conference on Ubiquitous Computing (Ubicomp). Virtual, September 2020.
- **Wyatt Lindquist**, Sumi Helal, Ahmed Khaled, and Wesley Hutchinson. 2019. IoTility: Architectural Requirements for Enabling Health IoT Ecosystems. IEEE Transactions on Emerging Topics in Computing.
- **Wyatt Lindquist**, Ahmed Khaled, and Sumi Helal. 2020. IoT-DDL: Device Description Language for a Programmable IoT. International Conference on Smart and Sustainable Technologies. Virtual, September 2020.
- **Wyatt Lindquist** and Sumi Helal. 2020. Requirements and Evaluation of a High IoTility Health Device Ecosystem. Manuscript in preparation.
- Ahmed Khaled, **Wyatt Lindquist**, and Sumi Helal. 2018. DIY Health IoT Apps. Demo Paper. Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SenSys '18), pp. 406-407. Shenzhen, China, November 2018.
- Ahmed Khaled, Abdelsalam Helal, **Wyatt Lindquist**, and Choonhwa Lee. 2018. IoT-DDL—Device Description Language for the "T" in IoT. IEEE Access, vol. 6, pp. 24048-24063.
- Ahmed E. Khaled, **Wyatt Lindquist** and Sumi Helal. 2018. "Service-Relationship Programming Framework for the Social IoT." Open Journal of Internet of Things, vol. 4 issue 1, pp. 35-53.
- Abdelsalam Helal, Ahmed Khaled, and **Wyatt Lindquist**. 2019. The Importance of Being Thing Or the Trivial Role of Powering Serious IoT Scenarios. Proceedings of the 39th IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 1852-1859. Dallas, Texas, USA, July 2019.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Looking towards the future, diseases and health issues that are commonplace today may soon become issues considered only in the past tense [1]. Changes towards this end will likely be led by an increased focus on individual well-being and health maintenance—shifting away from more traditional "reactive" intervention and sick care [1]. Such individual and social factors are already shown to play a significant role in one's overall health, accounting for about 60% of a given outcome (compared to 11% by medical care and services) [2]. These future transformations are motivated by a variety of factors, such as economics; emphasis on prevention may lead to decreased healthcare spending as issues are caught before hospital care is needed. One force in particular, however, plays a central role in this vision: technology.

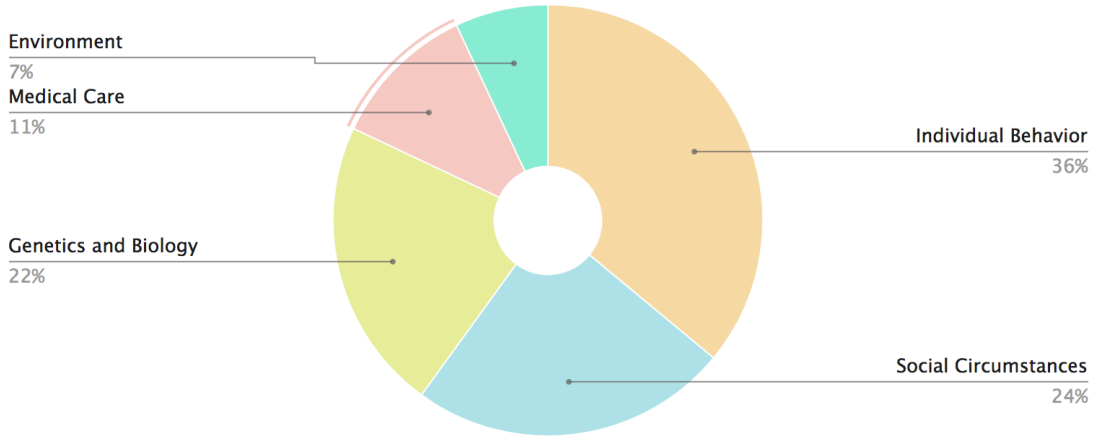

Figure 1-1: The Determinants of Health, a breakdown of health outcome factors [2].

Personal health devices and wearables are one set of key technologies that are anticipated to empower this transformation. These devices, along with a variety of connectivity features and services, are building the *Health Internet of Things (Health IoT)*—bringing with them new forms of streamlined data collection and simplified health monitoring. With an ever-increasing number of cheap and easy-to-use devices available, patients are more engaged in their own health and well-being than ever before. Further connecting these devices and their data to healthcare delivery will be

a major step towards a continuum-of-care system focused on detection and prevention, compared to today's treatment-focused point-of-care systems.

While IoT and wearable technology carries this potential, one must first consider if today's offerings are ready to meet such goals. Many of these wellness devices have grown alongside the personal IoT domain, taking inspiration from its developments and successes—for example, many traditional medical tools have joined the health IoT through the addition of connectivity features and cloud-based services similar to those of consumer applications (such as home automation). While these features empower the device's ease of use and the patient's access to data, they may not be sufficient to satisfy the full vision of health IoT: these ecosystems instead must be architected to meet new and unique requirements brought about by this vision before a device can truly fulfill its role in future healthcare scenarios.

1.1 MOTIVATION

At first glance, connected health devices may look and even act similar to their counterparts in other IoT domains; however, they also bring unique challenges inherent to the health and wellness ecosystem. When considering the future of healthcare as described above, various questions must be raised in regard to the functionality of an ideal health *thing*:

- *How can one ensure a health device is used properly?*
- *How can health data be validated and shared with healthcare professionals?*
- *How can health IoT ecosystems support a variety of IoT vendors and devices?*
- *How can health* things *interact with each other and the user to provide useful information and functionality?*
- *How can users be empowered in managing their collection of health IoT* things*, apps, and data more readily?*

These questions are only a sample of the many that must be answered as health IoT devices evolve and become commonplace. Analyzed from the perspective of the *thing* and its software, such concerns frame a set of overarching features and requirements that are especially relevant to effective health IoT ecosystems and their overall vision presented above. The following subsections expand on these questions and requirements, considering them from an architectural standpoint and positioning them amongst the motivations of this thesis.

1.1.1 Specialized Device Behavior

In the most basic sense, a *thing* represents some device or object made "smart" through additional hardware or software features and a networked connection. Such a connection allows *things* to easily share information with each other, the greater network, or the user. While a simple service offering data may be enough for some devices, health *things* often present additional usage concerns; attributes such as accuracy and reliability are critical when used in a diagnosis or treatment scenario. Human factors also play a role as more advanced medical devices are introduced into the personal health market—the device must be used correctly (at the right time, in the right manner, etc.) to be effective. Finally, such devices present new security and privacy concerns as they generate and handle protected health information.

In these scenarios, a *thing* architecture acts as the final line of defense in managing specialized behaviors. An effective architecture must provide facilities to handle the safety, privacy, and user engagement aspects of the device it runs on: delegating these tasks to IoT applications leaves too many opportunities for error. Especially considering the number of devices intended for use by inexperienced end users, the *thing* architecture must not only make the devices easier to operate correctly, but also harder to use in a wrong or unsafe manner. Catering to these specialized internal needs will play a large role in enabling effective IoT interactions.

1.1.2 The Role of Thing Architectures

While a specialized health device (as described in the previous subsection) may be built to behave as a connected smart *thing*, it is not necessarily prepared to interact in an *Internet of Things*. Consider that these *things* are unlikely to exist in a vacuum: they participate as part of a "smart space," surrounded by other devices that perform similar functionalities or offer useful information. A *thing* that only performs its actions on request can at best coexist; it cannot interact directly with other devices or utilize information from the rest of the system. While a developer may control interactions between devices manually or through an external service, such "outward" *thing* behaviors could automate or simplify this process through more direct and informed exchanges.

When considering questions such as "how can *things* interact (with each other)?", these outward features become critical to the health IoT domain. Within a smart space, a user's set of health *things* are often closely related: at minimum, these devices share a primary goal of improving or managing the user's overall health. In practice, they may also be capable of collecting shared information, or utilizing unique data from their neighbors. Whereas individual health devices normally have access to only a small part of a user's overall well-being, direct interactions can enable these *things* to obtain a more holistic view and provide better-informed services to the user.

To meet this goal, an effective *thing* architecture must also consider higher-level "outward" interactions in an effort to better integrate with other *thing* devices and cloud services. This includes features such as sharing capabilities and services with the smart space, and providing interfaces to find or request behaviors from other devices. Such features form the basis for *meaningful interactions* between devices and lead to the creation of new IoT applications that empower the user. How these interactions are defined and enabled within the *thing* architecture, therefore, will directly dictate how a health device can participate in an *Internet of Things.*

1.1.3 Health IoT Ecosystem Issues

The variety of connected health devices available today highlight another prominent concern: interoperability. Many of these devices are tightly coupled to their manufacturer's ecosystem, creating "silos" of fragmented services that limit how the end user can bring their devices together. While such ecosystems may offer highly integrated services and features, their issues are especially apparent in health IoT, where different manufacturers specialize in different types of wellness devices. As their needs (and IoT applications) change and evolve, even the most loyal user is bound to encounter a device that is not offered within their chosen ecosystem. In this case, the separated "silo" ecosystems are likely to limit the device overall.

Similar concerns are also present when considering the influence of *thing*-like objects such as companion mobile applications. Although not directly equivalent to a hardware *thing*, these apps are very common in health IoT and are often the primary point of interaction with their associated medical devices, as a companion app provides a very flexible interface (compared to that of the hardware *thing*). These apps, however, are also very sensitive to interoperability issues: owning a variety of devices may also mean managing each of their companion applications, which may be confusing or annoying for the user.

Interfacing with a variety of health IoT vendors and devices is another important component of an effective *thing* architecture. Such an architecture should understand various health data and device standards and work towards consolidating the multitude of ecosystems and silos present in a smart space, providing support and features that are appealing for vendors as well as developers and end users. Additionally, treating companion apps as first-class citizens in health IoT will also play a role in enabling meaningful interactions between the patient and their devices. Overall, focusing on interoperability will be a key step in solving various health ecosystem issues.

1.2 Problem Statement

Health IoT continues to develop rapidly while presenting a variety of unique challenges and concerns when compared to closely related application domains. This thesis aims to examine these issues from an architectural standpoint, seeking to identify the features and capabilities an IoT *thing* architecture must provide to best support the needs of personal health and healthcare delivery scenarios. Once these features are identified, this thesis investigates their feasibility—addressing how such requirements may be implemented and evaluated within an IoT *thing* architecture—and whether their realization indeed enables new solutions with connected health devices in mind. These goals, alongside the questions raised in the initial motivation, are summarized in the following research questions:

- **RQ1:** *What needs and priorities separate health IoT from other IoT domains?*
- **RQ2:** *What distinct capabilities and software features must a health IoT device provide to best support the specialized needs of personal health and healthcare delivery integration scenarios?*
- **RQ3:** *How can a device architecture meet performance and usability needs across a variety of health IoT scenarios and hardware?*
- **RQ4:** *What roles do various stakeholders play when using a specialized health IoT architecture in new applications?*

1.3 Thesis Scope and Structure

This thesis focuses on personal health and wellness IoT devices when identifying these unique needs. It aims to present a core set of requirements, such that each is believed to have a significant impact on the *thing* architecture's potential within these health devices. As these requirements cover a wide range of implementation features and domains, the implementation focuses mainly on concepts that facilitate flexible use and meaningful interactions between *things* and other devices.

On the other hand, this thesis does not consider *medical devices*—such as those found in a hospital environment or acquired through prescriptions—at the same level of detail, as their requirements are likely to differ substantially. This thesis also does not attempt to present a complete set of personal health requirements; for example, new or extended requirements may continue to be identified through additional evaluation or in consideration of specialized scenarios. Finally, this thesis does not focus on some of the core requirements involving additional and specialized research areas, such as security or incentivization.

The remainder of the document is structured as follows. Chapter 2 discusses the requisite background for this research and presents a set of relevant literature. Chapter 3 introduces the identified health requirements, along with the methodology used. Chapter 4 introduces an overview of the developed *thing* architecture, presenting an overview of the individual major components. Chapter 5 revisits these components, describing their implementation and functionality in detail. Chapter 6 then presents a set of experiments evaluating these components. Finally, chapter 7 concludes the thesis.

CHAPTER 2: LITERATURE REVIEW

An IoT *thing* architecture plays an important role in many facets of the greater Internet of Things. An effective architecture facilitates the developer to safely and easily interface with the hardware features of a device, dictates how the *thing* interacts with its surrounding smart space and users, and prepares the device to fulfill its requirements as part of a larger IoT platform or service. Each of these facets has received attention from a variety of research directions (such as data science or security) and application areas (such as smart homes or wearables), resulting in a vast number of new concepts and ideas. As connected medical devices and services become increasingly commonplace, the same holds true for health-related IoT, which is referred to as health IoT.

This chapter surveys work relevant to the requirements of health IoT and their potential implications in IoT *thing* architectures. Section 2.1 begins by considering developments in present-day health IoT ecosystems, in both commercial and experimental capacities. Section 2.2 extends this discussion, analyzing the direction of future health IoT ecosystems and their potential effect on future *thing* architectures. Section 2.3 narrows its focus to work on generalized *thing* architectures and their enabling features. Finally, section 2.4 discusses some initial work towards specialized health *thing* architectures, along with areas and issues that require additional efforts to be put forth.

2.1 EXISTING HEALTH IOT ECOSYSTEMS

What makes up an *IoT ecosystem?* In this context, an ecosystem refers to the complete set of components that work together towards some common goal. This includes device hardware (the *things* themselves), the network between these *things* as well as the functionality they offer, and the features of the cloud, IoT platform, or other services (such as mobile apps) connecting them together. An IoT ecosystem also involves a variety of stakeholders, including vendors who own the ecosystem,

third party developers who create new applications and interactions (depending on the openness of the ecosystem), and end users who take advantages of these features. Together, these stakeholders define the unique needs and expectations for each component of a specific ecosystem.

The whole of such an ecosystem is greater than the sum of its parts, especially when considering health IoT. While *thing* devices collecting data and performing actions make up the core of an ecosystem, their interactions with other components—such as mobile apps and cloud services—are what truly determine how a user can utilize the system and achieve their personal health goals. This highlights the importance of the *thing* architecture within the ecosystem: in addition to controlling the devices, the architecture plays a major role in defining and enabling these interactions. Although some ecosystems (especially those with tight control over the "silo" of available devices and features) may depend less on its flexibility, the *thing* architecture represents a common point that each ecosystem component must interact with in some capacity.

In the same manner, the needs of the ecosystem also influence the design of the *thing* architecture and facilitate its advancement. Changes and new ideas within the ecosystem promote evolution of the architecture as well, leading to a cycle of improvement that continues to push both forward. With this in mind, an analysis of present-day ecosystems, their challenges, and their direction can provide insight into the needs and requirements of future health IoT architectures. To illustrate this potential, the following subsections introduce some existing ecosystems and concepts with this capacity to drive improvement.

2.1.1 Commercial Ecosystems

The increasing popularity and availability of wearables and other connected health devices has caught the interest of vendors and manufacturers, resulting in a variety of commercial health IoT ecosystems. Most of these target the domain of personal health—offering various devices for home use and diagnosis—but have also begun to look towards additional use cases, such as connecting users and data with healthcare

providers. However, while such ecosystems may offer highly integrated devices and services, they are often tightly coupled: leaving less room for interaction with other ecosystems outside of their designated "silo," and limiting the types of devices that can be used in a single ecosystem.

iHealth Labs [3] offers health *things* such as pulse oximeters, body weight scales, blood pressure monitors, and thermometers, most of which communicate using Bluetooth and the user's smartphone. Their *MyVitals* app is capable of controlling these devices, taking measurements for storage and tracking within a proprietary cloud service. Withings [4] (previously known as Nokia Health) is another vendor offering connected health devices, including smart watches, body weight scales, thermometers, and sleep sensors. These devices send their readings over Bluetooth or WiFi to the Withings cloud service (or a custom solution for customers using their *Med Pro* service), which can be reviewed through the fitness- and wellness-tracking *Health Mate* app. Both companies have also introduced additional integrations with healthcare providers, such as remote patient monitoring and telehealth functionality.

Libelium's MySignals [5] ecosystem centers around an Arduino-based development platform capable of interfacing with a variety of MySignals-branded devices, including spirometers, temperature sensors, glucose monitors, and EKG electrodes. The platform is also capable of interfacing with a mobile app and proprietary cloud service for collecting and recording data. Unlike the two ecosystems mentioned above, however, MySignals offers additional flexibility through prototyping and modification, allowing new sensors to be integrated with the main device (although these integrations must be programmed and configured manually by individual developers).

Apple Health [6] and Google Fit [7] focus less on devices (although both offer their respective smart watches) and more on the software and cloud services needed in a health ecosystem. Rather than offer a set of first-party *things*, the platforms focus on enabling third-party integrations that allow users to collect data from multiple ecosystems in a single place. This is achieved through common APIs that

vendors can use within their own apps to read and write shared health data. Additionally, these services offer compatibility with various electronic health record standards and simplified integration with healthcare systems. Still, the onus is on individual vendors to support these platforms before such interoperability features can be taken advantage of.

2.1.2 Device and Data Standards

Although some health IoT ecosystems may be completely proprietary, many offer at least some support for various health-related standards. For example, a device standard may define a common API for *things* and their interactions, while a data standard helps simplify interoperability between platforms and service providers. The existence of these standards influences the design of an ecosystem and even plays a role in its effectiveness and adoption; commercial ecosystems can often use their standards-compliance as a selling point. This subsection introduces some standards targeting connected health devices.

The Continua Design Guidelines [8] (now part of the Personal Connected Health Alliance [9]) provide a set of health device standards alongside an open implementation focused on device connectivity and interoperability. The project includes guidelines for Bluetooth Low Energy (BLE), Zigbee, and USB connectivity, and builds on other standards such as ISO/IEEE 11073 [10] and the BLE Health Device Profiles [11]. Connected health devices may become “Continua Certified” (such as various A&D Medical [12] devices), assuring end users of their compliance and interoperability with similar devices. Overall, the standard provides a variety of guidelines, testing tools, and software examples to empower vendors and developers in creating interoperable health *things* and ecosystems.

Fast Healthcare Interoperability Resources (FHIR) [13] is a data standard facilitating interaction with electronic health records (EHR) and healthcare providers. The project, aiming to supersede and simplify previous Health Level 7 (HL7) standards, represents an EHR through a set of *resources*, holding reusable data types, metadata, and human-readable content. The standard is already used in a variety of

existing ecosystems, such as Apple Health and Google Fit (described above); however, it still must be supported by both the EHR vendor and the user's healthcare provider before it can be taken advantage of. The standard has also seen some integration with decentralized storage platforms such as *Solid* [14] and *MyData* [15], empowering users to better access and "own their data" in similar scenarios.

2.1.3 Health Ecosystem Research

Connected health devices have also seen use in various research-based health IoT ecosystems. These systems often target an individual health issue or scenario (such as diabetes), which is used to define the specific requirements and challenges the architecture aims to solve. Implementation focus in these ecosystems varies between low-level device features, *thing* interaction and connectivity features, and cloud-based features depending on the needs of the target scenario. Still, many of these identify challenges that are applicable to a variety of health IoT situations. This subsection introduces some of these specialized health IoT ecosystems.

Harous et al. [16] introduce MA4OCMP, an ecosystem targeting obesity management. The system collects data from a combination of health sensors and social media activity, which are stored, analyzed, and shared with healthcare providers through a cloud service that provides customized suggestions, warnings, and recommendations to at-risk users. The project focuses on the use of multiple data sources and the interaction between patients and healthcare providers in effectively managing obesity. In the same vein, Deshkar et al. [17] analyze diabetes ecosystems, making similar findings on the importance of context-awareness between the various stakeholders within the ecosystem. Deshkar also identifies a common focus on interoperability, security, and privacy amongst these systems.

Debauche et al. [18] present an ecosystem for patient and elderly monitoring using a fog IoT architecture. This system uses a smart gateway edge service in addition to a sensor network and cloud architecture to track state and behavior changes of patients. The project focuses on data reliability—storing and managing data on the edge device if the cloud service is unavailable—and privacy, ensuring

GDPR conformance as information is passed between the sensors, edge device, and cloud service. The ecosystem again focuses on interaction with stakeholders, sending alerts and warnings to healthcare providers and allowing them to validate abnormalities through a graphical web interface.

Some concerns and challenges are ubiquitous across these ecosystems; for example, security and privacy remain a prominent concern when sending protected health data and information to the cloud. As such, there exists a variety of research projects and solutions targeting these facets in the context of health IoT ecosystems. Lomotey et al. [19] propose a system managing data traceability and provenance in wearable devices. Verifying and associating data sources with the correct patient is critical in a variety of scenarios, such as adding to an EHR. Mahalle et al. [20] present an access control system that calculate trusted value based on parameters captured from the smart space. These trust values can be used to manage user identity across *thing* devices. O'Donoghue et al. [21] and Yang et al. [22] focus on data validation, consistency, and reliability across connected sensors, wearables, and mobile apps.

2.2 Future Health IoT Ecosystems

While present-day health ecosystems are already enabling a variety of new and useful interactions with connected health devices, visions of future ecosystems continue to evolve. These future concepts provide additional insight into the current needs and problems of existing ecosystems, and can lead the direction of *thing* architecture development as previously described; future *thing* architectures cannot base themselves on only the needs of the present-day. This section introduces works considering future ecosystems in this manner, as well as works that identify and analyze the specific health IoT challenges they must solve.

Deloitte Insights [1] envisions an increasing focus on the consumer in future health ecosystems, with greater importance given to well-being and maintaining health rather than responding to illness. Alongside this, health sensors will become increasingly integrated and ubiquitous, providing a large amount of personal health

data for early detection of various health issues. Empowered by this information, end users will be better informed to determine when and how they interact with healthcare providers—for example, more routine health issues could be handled at home without direct intervention from a physician.

Deloitte argues that interoperability and open platforms will be critical to the effective use of this abundant health data. As much basic medical care is relatively algorithmic, access to this data will play a large role in proactive intervention through artificial intelligence, therapeutic and health algorithms, and other new combinations of services provided by vendors and other stakeholders. While many of these interactions are likely to take place in the cloud, the *thing* architecture still plays an important role in collecting and channeling the data that is needed to enable this vision. Similarly, interoperability will be needed not only at the ecosystem level, but the architecture level as well.

However, considering data collection and interoperability alone is a relatively narrow vision; as ecosystems become increasingly ambitious, their potential needs and requirements grow as well. In addition to efficiently sharing data, future ecosystems may also require new ways to interact with the user, create new application opportunities, manage uncertainty, and much more. In anticipation of these new ambitions, a variety of potential challenges have been identified and considered by researchers. Each of these challenges may play a role in the design of health IoT ecosystems and *thing* architectures, in the same manner as interoperability is envisioned to.

Laplante et al. [23] investigate a set of case studies, such as dementia and patient safety, to determine potential quality issues and needs. The authors identify a variety of privacy and safety requirements and relate them to an overarching "caring" requirement. Caring is defined here as a combination of qualities such as reliability, trust, and empathetic interactions. These caring requirements and expectations are mentioned to vary between individuals and scenarios; the sub-qualities that make up caring may change depending on the needs of the

stakeholders. The authors suggest that collaboration with various healthcare workers and providers will be critical in identifying the specifics of these requirements.

Farahani et al. [24] analyze health challenges from the perspective of their fog-driven IoT architecture. Five major challenges are identified: 1) data management, where the volume and variety of data makes standardized processing difficult, 2) scalability, from individual users to hospital deployments, 3) interoperability and standardization, especially involving regulatory concerns, 4) interfaces and human factors, due to patients (especially elderly) that may not be familiar with connected devices, and 5) security and privacy of protected health data. The authors also present a case study to illustrate these challenges further.

Plaza et al. [25] identify similar requirements when considering the needs of healthcare IoT platforms and middleware, again referring to scalability, interoperability, and security and privacy. Amongst these needs, the authors also identify dynamic discovery and reconfiguration as important features of a health IoT architecture. Such discovery is critical due to the number and heterogeneity of sensors that may present in a health system, while reconfiguration can help devise adapt to changes without interfering with the operations of other devices. The authors then compare how these requirements manifest in a variety of present-day health ecosystems and architectures.

2.3 Generalized Thing Architectures

Zooming in on the *thing* architecture, one specialized for health IoT will likely share a variety of similarities with its more generalized personal IoT counterparts. In addition to abstracting hardware and other internal features, *thing* architectures also aim to enable outward interactions and integrate with other *thing* devices and cloud services. This includes features such as sharing capabilities and services with the smart space, and providing interfaces allowing other devices to use these capabilities. Such features help enable a device or platform as an IoT *thing*, rather

than just a connected device. This section introduces some explicit IoT architectures, along with concepts that are likely to play a role in specialized health IoT scenarios.

ARM Mbed [26] presents an architecture focusing on the interaction between a *thing* device and the cloud platform, where device functionalities can be linked and re-exposed as cloud services that can be provisioned, discovered, and managed through a set of standard interfaces. The architecture runs on Cortex-M devices using its Mbed OS, which provides a lightweight library for interacting with hardware, communication protocols, and network security. Applications can also be programmed and compiled through a web-based IDE, minimizing the setup requirements placed on the developer. However, the architecture is tied closely to the ARM ecosystem, both its cloud service and microprocessor models.

The Network of Things (NoT) [27], introduced by the National Institute of Standards and Technology (NIST), proposed a foundational design for IoT systems. NoT breaks down IoT systems into a set of four underlying concepts that influence its design: *sensing, computing, communication,* and *actuation.* The work also introduces a set of primitives and elements to describe the functionality of individual sensors and groups of devices, while also taking into account issues such as trust, privacy, and security. The concepts presented by NoT can directly map to the implementation of lower-level architectures and designs in a variety of IoT domains.

MOSDEN [28] is an IoT middleware targeting mobile devices, allowing users to collect and analyze sensor data through a service model. Users can connect new sensor types without the need to directly program such an interaction; this is achieved through a plugin architecture, where plugins (developed by third parties) can be added and removed on the fly (downloaded from the smartphone's application store) to support a specific sensor type or brand. A specialized mobile app allows users the view detected sensors and the data that has been collected. Smartphones may play similar roles in health IoT systems, as they are a primary mode of interaction for many connected health devices. However, in this case, the smartphone acts as an endpoint (interfacing with only sensors) and cannot easily participate in *thing* interactions.

2.3.1 Programming Models

As mentioned above, interactions between *things* play an important role in all forms of *thing* architecture. Programming models allow developers to more easily create these interactions, facilitating new connections between *things* and services. These models are important in health IoT architectures as well, especially given that health sensors are frequently used in groups and collect related measurements. This subsection presents a selection of personal IoT programming models that can facilitate these kinds of interactions.

If This Then That (IFTTT) [29] [30] is a web-based service that allows users to manually connect various internet-based services and applications through trigger-action rules: if a certain event occurs, then perform some action. IFTTT can utilize various cloud APIs offered by service vendors, alongside limited support for smart home products and Android system functions. These interactions form *applets*, which may be defined manually or downloaded from other users. In a similar vein, Yun et al. [31] developed Things Talk to Each Other (TTEO), which focuses on if-then rules specifically targeting *thing* devices. The TTEO architecture consists of two platforms, one which discovers and communicates with the *thing* devices, and another which manages the logic and execution of the trigger-action rules. Users can add new rules through a specialized mobile app, again highlighting the importance of smartphones in an IoT ecosystem. However, these approaches also require the user to manually identify their desired behaviors, which may be less practical in the presence of many devices (such as in a future health IoT ecosystem).

The Social Internet of Things (SIoT) [32] [33] mimics human social networks with the properties of smart *things.* The authors present an architecture defining different social relationships that describe how devices may interact, namely: parental (*things* built by the same vendor), co-location and co-work (*things* existing in the same place or providing similar functions), and owner (*things* owned by the same user). These relationships can then be used to detect or suggest more relevant

interactions between *things*, rather than requiring explicit user consideration. Similar social concepts play a role in the Atlas Thing Architecture (described below).

2.3.2 Thing Descriptions

Understanding the features and capabilities of a *thing* also plays a role in structuring interactions within an IoT ecosystem. However, the wide landscape of (health) ecosystems on offer has resulted in a significant amount of fragmentation; even a device that supports new and exciting interactions may be limited by its programming interface or cloud connection. Such platforms offer these synergistic features, but lack the provisioning needed for intercommunication. A common, base layer of compatibility that reflects a device's capability and can be understood by other *things* has the potential to overcome these issues.

The Web of Things (WoT) framework [34] [35], introduced by the World Wide Web Consortium (W3C), aims to represent thing capabilities through web-based services, described and accessed with a semantic WoT Thing Description (WoT-TD). The TD data model can be extended with domain-specific information and is represented in a serialized JSON format. However, while the TD is capable of describing the properties of individual *things*, additional information is required to represent potential interactions or relationships between *things*. These concepts are extended in the WoT Asset Description (WoT-AD) by Le et al. [36], which groups physical *thing* devices into "Assets" that provide a set of services. This JSON-formatted description defines high-level interactions, allowing multiple *thing* APIs to be provisioned through a single functionality-focused description. While this simplifies device setup, however, its focus on the groups of *things* also makes it difficult to define more generic descriptions.

The Atlas Device Description Language (DDL) [37] [38] is a human- and machine-readable device description that aims to simplify the integration of *things* and services in a smart space. The XML-based format defines metadata and functional descriptions for the various sensors and actuators that may be part of a *thing*. DDL descriptions are meant to be developed initially by device manufacturers

and OEMs, reducing the responsibilities on developers integrating new devices. The DDL forms the base of Khaled's Internet of Things DDL (IoT-DDL) [39], which describes additional metadata and capabilities to better facilitate interactions between *things*. The IoT-DDL is extended further in this thesis, in response to some of the health IoT requirements identified in chapter 3.

2.3.3 The Atlas Thing Architecture

While many architectures focus on communication through a single point (such as a cloud or edge device), the Atlas Thing Architecture [39] [40] focuses on the potential derived from direct *thing*-to-*thing* interactions. The architecture is based on the original Atlas Sensor Platform [41], building on the Service Oriented Device Architecture (SODA) [42] and abstracting the device's hardware to provide new *thing* functionalities through the DDL project discussed above. The Atlas platform focuses on self-discovery and advertisement of device characteristics through the DDL, interaction with remote provisioning and management, and secure communication with other devices in the smart space.

The Atlas Thing Architecture builds on top of this platform, introducing new features such as the extended IoT-DDL and improved interoperable communication [43]. The architecture also introduces an inter-*thing* relationships framework [44] that defines a set of relationships to logically and functionally describe new connection between unrelated services and devices in a smart space. Such relationships can be specified manually or inferred to enable new *thing* interactions. Like the IoT-DDL, this thesis expands upon the Atlas Thing Architecture in response to the health IoT requirements identified in chapter 3.

2.4 Health IoT Architectures

While generalized IoT architectures may provide features that are useful in health IoT contexts as well, a specialized health *thing* architecture can better account for the unique needs of healthcare scenarios. Even if such a feature is not exclusive to

the health domain, it may come with different priority and importance compared to other domains such as personal IoT. This section discusses the implementation of some specialized health IoT systems and architectures.

Azimi et al. [45] introduce HiCH, a fog-based health IoT architecture that is divided into two components; the distributed *thing* architecture, and a higher-level cloud component. At the *thing* level, the architecture focuses on requirements such as data accuracy, availability, and security. The *thing* also performs analysis on the data it collects in an attempt to lower the amount of needed communication with the cloud service. In the cloud, the architecture utilizes machine learning and other analytics to detect trends and issues in health data. While the architecture fulfills some of the lower-level needs of *thing* devices, it focuses mainly on sensor data collection, and does not fully consider the potential for interaction between *things.*

Catarinucci et al. [46] offer another architecture targeting healthcare systems, called the smart hospital system (SHS). The authors utilize a variety of communication protocols to collect and monitor data, putting much focus on the interoperability features of these devices and networks. This data can then be accessed uniformly by healthcare providers through a cloud service, or monitored to send push notifications to caregivers on critical sensor events. Various services for stakeholders are exposed through web and mobile applications, which consider their own security and privacy challenges. Again, this architecture does not fully consider the full range of potential health IoT requirements, such as device safety or inter-*thing* interactions.

Banos et al. [47] describe a framework and architecture for mobile health (mHealth) applications, and present an implementation for mobile devices called mHealthDroid. The framework consists of a set of layers designed to support new mHealth applications, including support for communicating with biomedical devices, storing and analyzing health data, and displaying information to the end user. The framework also describes “service enabler” components to abstract events such as alerts and notifications within the mHealth application. The Android architecture implements a portion of the framework, defining various adapters for

communicating with wearables, as well as a graphical interface for displaying sensor data. The framework places focus on the role of the developer in an mHealth ecosystem. However, beyond Android apps, the framework cannot be used directly on physical devices such as health sensors.

FIWARE [48] is a cloud platform developed under the European Commission to facilitate the creation and adoption of new cloud services. While the platform itself does not specifically target health, the *Future Internet CHallenge eHealth* (FICHE) accelerator program focused on its use in e-health applications. Celesti et al. [49] describe the creation of a health system and application using this platform. The system consists of a set of FIWARE components called "generic enablers" (GEs) that facilitate collecting and storing data, implementing service endpoints, and managing devices. Data can be collected from heterogenous devices and uses the Open Mobile Alliance (OMA) standard for transmitting information. In addition to the core components, the system also offers GEs for developers to create front-end applications, as well as widgets that end users can compose. However, while the system fulfills most of its needs through the cloud, it does not consider potential contributions or interactions from the devices themselves.

2.5 SUMMARY

This chapter presented a breadth of related works covering concepts related to digital health and IoT architectures. The design of a new specialized health architecture is likely to be influenced by a variety of factors, including present-day medical devices and existing standards, design patterns, and specialized applications. The needs of future health scenarios and even existing personal IoT architectures are also likely to play a role in the design of a new health IoT; the sections of this chapter elaborated on each these considerations.

In terms of personal health, many existing devices stem from commercial products targeting end users, as presented in subsection 2.1.1. Such devices are usually tied to a proprietary cloud service, resulting in more closed-off systems that

focus mainly on integration with other products from the same company. Still, some of these devices have also brought about new standards that promote interoperable data and ease of use beyond their local ecosystems. Some of these standards and related efforts were discussed in subsection 2.1.2.

Personal health also shares a close relation to traditional personal IoT; as such, it is likely that some needs and solutions would carry over. Section 2.3 discussed concepts relating to more generalized thing architectures, such as programming models and device description languages, as well as the Atlas Thing architecture that this thesis builds on top of. Although not directly health-related, these features are likely to correspond to the solutions presented in the explicit health IoT architectures discussed in section 2.4.

Table 2-1: Related works comparison.

	Primary Use Case	**Architectural Layers/Focus**	**Full Device Architecture**	**Inter-*Thing* Interactions**	**End-User Developers**
HiCH	Hospital Environment	Fog/Cloud Data Analysis	No	No	No
SHS	Hospital Environment	Sensor/Network Hardware	Contiki OS	No	No
mHealthDroid	Personal & Home Use	Android Apps & Paired Sensors	Partial (Mobile Only)	Partial (Mobile Only)	Yes (App Services)
FIWARE	Remote Hospital Care	Cloud Services	No	No	Yes (Custom Widgets)
Atlas Health	Personal & Home Use	Device-Local Needs & Utility	Yes	Yes	Yes (IDE, DDL, etc.)

These health architectures begin to answer some of the future needs presented in section 2.2. Although more generalized than the solutions discussed in subsection 2.1.3, these architectures still vary in their specific goals and intended use cases. Table 2-1 shows a comparison of these architectures, alongside the solutions presented in this thesis. One main point of comparison amongst these related works is their primary use case: many focus on scenarios such as hospital care and patient monitoring. Amongst the works above, only mHealthDroid shares a personal health target (that is, cases where healthcare professionals are necessarily involved).

The table also shows various differences in the architectural design and focus of each project. Partially due to their primary use case, many of the architectures put emphasis on the structure of their fog/cloud layers (along with their networking), using the *thing* devices more as simple sensors meant to collect and send data for further processing. The table expands on this further in the *Full Device Architecture* column, which considers the actual software running on devices. For example, HiCH and FIWARE do not specify an explicit device architecture, while SHS uses Contiki OS [50], a general purpose IoT middleware. Again, mHealthDroid offers the closest to a detailed device architecture, although it only targets mobile devices. Such an architecture is a primary component of the work in this thesis (described further in chapter 4), encompassing concerns such as direct interactions between *thing* devices (without using a cloud, fog, or gateway); amongst the compared works, similar features are not considered in the same depth.

Interactions with stakeholders, such as developers (those who program a device but do not work for the vendor) are also a main point in this thesis—discussed throughout chapter 5—resulting in a variety of tools such as an IDE plugin and other graphical tools. While some of the other research systems are more rigid, some do present similar features: mHealthDroid services can be used in new mobile apps, and FIWARE allows developers and even end users to create “widgets” using its core functionality. Overall, though, this thesis and Atlas Health focus on a set of features in personal health that have received less attention in similar works.

CHAPTER 3: REQUIREMENTS FOR HEALTH IOT

As discussed in Chapter 1, the devices within a Health IoT ecosystem each have specific needs and unique functionalities that are either less likely to be found or less of a focus in other application domains. With the variety of emerging health IoT systems and devices in a highly fragmented and evolving market, special consideration must be taken to these unique needs: how can they be met fully, safely, and consistently using the functionalities available within a device and its surrounding smart space? This is the role of a specialized *thing architecture*; rather than offer its features in isolation, a focused architecture would help a device fully utilize its potential collective and safe use. Such an architecture plays a large role in guiding how a health IoT device performs its functions and interacts with the user and other *things*, features which are critical to a powerful and effective IoT system.

The unique needs inherent to the digital health domain are requirements that represent areas of common concern amongst health IoT systems. Each requirement presents a set of features that must be addressed within the *thing* architecture to maximize the utility of a connected health device. These requirements are proposed from the perspective of the *thing*; they do not map directly to specific user needs or use cases, especially when considering the variety of health IoT application scenarios. Instead, a requirement helps structure the functionality of a *thing* architecture, providing core features that assist the device and developers in meeting these specialized needs of the end user, regardless of the specific scenario.

This chapter describes a methodology to seek out these specialized needs, and presents a set of requirements that have been identified as integral to the implementation of an effective health IoT architecture. Section 3.1 breaks down the methodology into four main steps. Section 3.2 describes the four main groups of requirements that have been established, along with how they fit in amongst the variety of health IoT devices. Finally, sections 3.3-3.6 describe each requirement in full detail.

3.1 METHODOLOGY

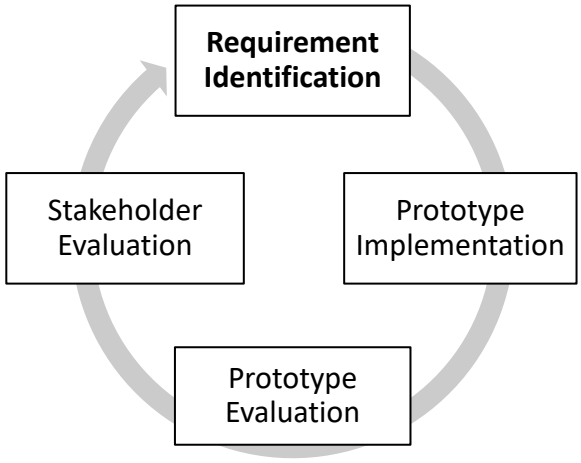

Figure 2-1: Stages of the chosen methodology, starting at the bold element.

Identifying, implementing, and evaluating these requirements necessitate a variety of approaches. This methodology is broken into four stages, illustrated in figure 3-1. The first step, *requirement identification,* uses a research-based approach to select potential requirements for further evaluation. This process of choosing requirements for a *thing* architecture calls for careful consideration regarding the needs of the devices themselves, as well as the needs of their users and the spaces they reside in. Identifying such "system" and "ecosystem" concerns is discussed in the following subsections. The second step, *prototype implementation*, and the third step, *prototype evaluation*, focus on the system side of these concerns; these steps determine the feasibility of a newly chosen requirement. Health devices vary greatly in hardware and intended use, making an experimental approach with prototypes and example scenarios valuable in finding limitations and evaluating performance characteristics. The fourth step, *stakeholder evaluation*, considers the ecosystem surrounding the architecture: a requirement may be further validated through surveys and other user-facing studies with stakeholders. This relationship between a stakeholder's needs and the features of the *thing* architecture is explored further in the following subsections. As these steps are followed, an iterative approach may be

taken; the requirements are likely to be closely related, allowing one to be further refined as others are identified and evaluated.

3.1.1 System and Ecosystem Needs in Health IoT

While a *thing* architecture directly influences how a device operates, it also provides the *thing* with the interfaces it needs to interact with the user and the devices of their smart space. This potential for interaction is critical within a successful IoT ecosystem; a well-designed system may make an effective medical device, but its capability for meaningful interactions makes it an effective medical *thing*. Therefore, the needs of the ecosystem are of equal importance to the needs of the system itself when considering potential requirements. As such, a variety of methods must be used to adequately research these needs, both system and ecosystem. This thesis makes use of literature review and market research in the first step of the methodology.

The literature review, reflected in Chapter 2, aims to reveal shared themes and patterns amongst various challenges and solutions within health IoT. A work may present a system in response to a specific problem, such as gathering and presenting data for obesity management or optimizing communication between wearable devices. Looking at these solutions across a variety of scenarios within the health domain reveals common ground amongst them, which can influence the specification of a new requirement. Other research using existing IoT architectures may reveal shortcomings in their use within the health IoT domain, that can be solved through a specialized architecture with its own set of requirements. Together, the literature review serves to identify the direction of future health IoT systems and the missing features that could enable them or enable them further.

On the other hand, market research aims to identify the trends and needs of commercial health IoT devices existing today. Looking at today's devices helps reveal the features that are popular, desired, or useful from the perspective of both the end user and the vendor. This information may reveal requirements applicable to a broader set of health IoT devices and provide insight as to how problems in the

health domain are being solved with existing products. Commercial devices are especially important when considering the ecosystem of health IoT; in many offerings, the device is only one part of the full product. Cloud features, companion mobile applications, and other services all play a prominent role in commercial health IoT ecosystems and, by extension, their device architectures. Staying aware of how these services augment and extend the "IoT features" of a connected medical device is another important component of identifying requirements. In this thesis, a variety of commercial personal health IoT devices were requisitioned, analyzed, and compared. In addition to influencing the requirement selection, some of the results of this work are presented in Chapter 6.

Comparing these methods also provides insight; the disparity between the envisioned scenarios of research works and the present state of commercial health devices may reveal deficits that influence or create requirements. In this manner, a certain level of intuition and opinion is also necessary when considering these methods. While grounded in the research directions described above, narrowing down the needs of personal health and investigating these disparities likely involves an aspect of prediction and personal judgement; looking to the future of health IoT devices and systems also plays a role in identifying requirements. Each requirement category presented in this chapter is derived from a combination of the methods described above; specific instances of influence will be cited in each section.

3.1.2 Stakeholder Roles

When considering a new health IoT *thing* architecture, a variety of stakeholders stand to benefit. This includes the *vendor*, who uses the architecture to build a new device and service, the *developer*, who takes advantage of the architecture (and the vendors devices) to create new interactions and applications within a smart space, and the *end user*, who chooses these devices and applications to meet their personal health goals. These groups may sometimes overlap (for example, the developer may also be the vendor, or even the end user), but each brings different needs and expectations from the *thing* architecture. Extending the previous subsection, system

needs may be more important to the vendor's implementation, while ecosystem needs are the focus in the developer's applications.

Satisfying the end user's needs is also critical to a *thing* architecture's success; however, these needs may not map directly to the specification of a new requirement. For example, when purchasing a connected health device, although the end user is interacting with *thing* architecture, they expect their specific needs to be met by the actual device or IoT application. For the user, the architecture provides a base set of features enabling these needs to be met, "behind the scenes." This is an important factor when identifying new requirements: end user needs may have to be considered from the perspective of the vendor or developer to determine how they may manifest at a lower level. The *thing* architecture itself can then provide a backend simplifying or enabling these features for the vendor and developer. Overall, when considering stakeholders, the needs of vendors, developers, and end users are likely to be closely related and codependent in terms of an architectural implementation, even when these needs do not map directly to the requirements of the *thing* architecture. In this thesis, such needs and considerations influence the design of various architectural features (described in chapter 4), as well as the experiments and metrics used to evaluate them (as shown in chapter 6).

3.1.3 Discussion

While this thesis presents what is believed to be a core set of health IoT requirements, it is important to consider the possibility that new or extended requirements continue to be identified. For example, further stakeholder evaluation in the iterative methodology described above may reveal additional needs that could be best answered through the introduction of a new requirement. This may be the case when considering specialized health scenarios, such as those involving users with disabilities. Such stakeholders may require accessibility that call for additional requirements, specifically targeting the domain in question. This thesis does not claim to provide a complete list of these health IoT requirements; rather, the selection presented is believed to have a significant impact on a *thing* architecture

maximizing its potential in utilizing new digital health devices, along with the interactions and applications they enable.

3.2 Choosing Architectural Requirements

When choosing the core set of architectural requirements, the entire landscape of health IoT must be considered. A single device can be placed along a spectrum depending on factors such as reliability, intended use (monitoring, diagnosis, treatment, etc.), and safety concerns. For example, the Food and Drug Administration (FDA) divides health devices into two categories: wellness and medical [47]. In this categorization, a given device is classified based on its potential risk to the user, as well as its claims towards specific conditions. Under these guidelines, a wellness device focuses on general lifestyle activities such as fitness and diet, while a medical device targets actual medical conditions and aims to diagnose, monitor, or treat the user.

The architectural needs and expectations of a device depend greatly on this classification, or position within the spectrum of health IoT. A medical device as described above, for example, may require secure transmission of measurements back to the healthcare provider's systems. However, enforcing such security and connectivity requirements on a wellness device may make less sense; unverified fitness information would be of comparatively little use to a healthcare provider, if considered at all. To account for these differences, this thesis defines a set of tiers to organize the variety of health IoT devices as follows:

- *Wellness*: Similar to the FDA definition above, these devices are intended for general use; their features would not be used for definitive diagnosis or treatment. This tier focuses on ease of use and often involves higher level information, such as a fitness band reporting steps taken or calories burned.
- *Personal Health*: As the health IoT ecosystem expands, so do the capabilities of devices available for personal or home use. This intermediate tier

encompasses devices that are capable of more comprehensive or accurate functionality. For example, while these devices may not be used or prescribed directly by healthcare professionals, their readings may influence the user to seek further diagnosis (such as those from an electrocardiogram sensor within a smart watch). This includes devices like blood glucose meters that are intended for diagnosis or treatment management but are not usually controlled or managed directly by healthcare professionals.

- *Medical*: At the highest tier, these devices are used directly for diagnosis and treatment by healthcare professionals, where incorrect use or erroneous readings can have a significant effect on the user. Such a device is likely to be prescribed to the user or owned by the healthcare provider, and is usually designed to operate without the support of additional devices (such as other things or the user's phone). One such example is a cardiac Holter monitor, usually prescribed by a doctor and worn for a period of time before being returned for analysis. Sometimes these devices even come with their own mobile device, to prevent the patient from needing or involving their own.

When defining a requirement, its properties can be positioned amongst these tiers to help clarify intentions and expectations within the health IoT ecosystem. This thesis organizes such requirements into four main groups (with variable relevance across the three device tiers). These requirements consider features needed to enrich interaction within devices (such as how a *thing* can expose its capabilities programmatically to application developers as well as cybernetically to the smart space) and with the user (by enabling an end user to properly and safely use a device), as well as their roles in each of the defined tiers.

This chapter covers four overarching requirement categories, each summarized below. Of these categories, requirements I and II are covered in the greatest detail and are the primary focus of this thesis. While all requirements play some role in an effective IoT architecture, each relates to a specific tier in which it has the greatest potential for impact. As this thesis primarily targets the wellness and

personal health tiers, Democratization and Device IoTility are likely to play the largest roles in terms of their influence on a *thing* architecture. This relationship between tiers and requirements is visualized in table 3-1. Still, the role of the other requirements cannot be discounted; each has the potential to influence the design of an IoT architecture at any tier. As such, requirements III and IV are still developed to a level that introduces the importance of their features, albeit in less detail. The four requirement categories are summarized below.

I. *Democratization*: This requirement focuses on the compatibility and interoperability of various health IoT devices. While vendor-specific services and support are often an important part of a device's offering, an over-reliance on these features can limit a device's effectiveness in the greater IoT ecosystem without some level of cross-compatibility and device ownership control. These features are most important at the wellness and personal health tiers, whereas a device in the medical tier is most likely to be owned by a health organization (a hospital or a clinic, for example) and for numerous reasons may not prioritize democratization.

II. *Device IoTility*: This requirement introduces several powerful features a health IoT device could offer by utilizing its connectivity with other *things*—such as another health device or the user's smart phone or smart TV—to create *meaningful interactions* between devices. Similar to democratization, this requirement is most important at the wellness and personal health tiers, while medical devices are more likely to be protectively self-contained in their functionality.

III. *Safe Use*: Within a health device, safe use refers to the ability to ensure correct and reliable input/output interactions with the user. This means an actuator *thing* should have limited ability to harm the user or its own hardware, while a sensor *thing* should have the capacity to detect erroneous readings and conditions. This requirement becomes increasingly important at

higher tiers (personal health and medical), where user-facing risk is higher and accurate functionality is critical.

IV. *User Identity & Privacy:* While security and privacy are a major concern across all connected devices, these features are exceptionally important in health IoT devices when the data must be generated by or shared with a healthcare provider or other third party and even complying with mandated regulations such as the Health Insurance Portability and Accountability Act (HIPAA) [48] and the General Data Protection Regulation (GDPR) [49]. Information from a device must be trusted and secure as it is transmitted between things, spaces, and ecosystems. This is an important requirement for both the medical and personal health tiers, where the sharing of data is more likely to involve healthcare providers and protected health information.

Table 3-1 shows the relationship between these four requirement categories and the three tiers presented earlier. Each requirement is ranked against its importance in each tier, represented by a number of checkmarks; for example, a single checkmark may imply only minimal focus is needed, while three checkmarks mean a requirement has a large influence within that tier. As shown in the table, the needs of the wellness and personal health tiers tend more towards Democratization and Device IoTility, while Safe Use and User Identity play larger roles in the medical tier. In the following subsections, further details are provided on these requirements.

Table 3-1: Significance of requirements per defined tier.

	Wellness	Personal Health	Medical
Democratization	✓✓	✓✓✓	✓
Device IoTility	✓✓	✓✓✓	✓
Safe Use	✓	✓✓	✓✓✓
User Identity & Privacy	✓	✓✓	✓✓✓

3.3 Requirement I: Democratization

In the context of health IoT architectures, the term democratization refers to a thing's capabilities and functionalities outside of its "silo:" the software ecosystem created and supported by its manufacturer or vendor. Many present-day IoT devices tend towards a pattern of closed systems with limited capacity for inter-silo functionality; while support for and interactions between *things* within the same silo may be extensive, compatibility with other devices is often limited to select manufacturers or nonexistent. These concerns extend across many aspects of the thing's software, from communication and data storage to the behavior of their relevant smartphone apps. For all of these aspects, a balance must be found to maximize the level of health IoT democratization without being too aggressive and undermining the business strategies of the vendor.

When considering such a balance, democratization is most important and most useful to the core functionalities of a health device. Features such as recording a measurement or configuring settings make up the lowest level of what could be considered "using" the device; at this level, the user should not have to worry about how a thing fits in with their existing digital health ecosystem. If each device or ecosystem requires, for example, its own companion smartphone app to function, a user with many things could quickly experience "app overload" [50], making it difficult to effectively use and manage these devices.

While many connected health devices such as fitness bands and other smart watches are expected to exist with a tightly-coupled companion app, health IoT has also introduced new connected variants of many traditional health devices where this is arguably not the case. *Things* such as blood pressure and heartbeat sensors look like their traditional counterparts, but may require additional apps, accounts, and ecosystems that are not immediately apparent. Consider a pulse oximeter *thing*, which has an integrated screen as well as a companion app. After purchasing the device, the user attempts to take a measurement but discovers the device must first be set up through the app; despite having the physical capabilities to function

immediately, the device is tied tightly to its silo (many present-day commercial devices, such as the *Withings* temporal thermometer [51], follow this pattern of requiring at least setup before independent use). On the other hand, a user's glucometer (e.g., the *iHealth Gluco+* meter [52]) can take readings even if the companion app is never installed or set up. While companion apps often provide a variety of additional features and conveniences, they should not be required if the user only needs basic functionality or can use another app of their choice (such as one they know well and are accustomed to).

3.3.1 Shared APIs

Considering how these companion apps communicate with their devices highlights another aspect critical to true health IoT democratization: shared APIs. Even if a device's hardware can be operated independently, it cannot act as a *thing* without connectivity to a smart space or other smart devices. Of course, eliminating these silos through a single standardized API is unlikely; vendors will always want to prioritize interactions between their own devices. However, even a limited subset of API compatibility, such as that defined by the Personal Connected Health Alliance (previously, the Continua Alliance) guidelines [9], could greatly improve a *thing's* ability to interact with its smart space. Imagine a patient with a low-power, connected body temperature sensor. Such a device is meant to be attached directly to the body, meaning it likely has little in terms of a physical interface (such as no integrated display to show the current temperature), and limited capacity for physical interaction (such as a sole button for toggling power).

The temperature readings and any needed configuration are instead exposed through a companion mobile app. Unlike the previous two examples, however, the device cannot be used properly without the app, even though the hardware itself is functioning. This could be problematic in certain cases where the app cannot be used; imagine a patient with disabilities and an unsupported, specialized device. Luckily, in this case, the device exposes a minimal standard API with basic functionality. Perhaps the user holds their mobile device near the sensor and

receives usage instructions as well as the ability to perform a basic read from the sensor (thereby receiving their body temperature as desired). More advanced configuration features are not exposed through this API, but the user is still able to use the device, despite using a different app.

Introducing more open APIs, however, does have implications regarding the safety and security of these smart devices [53]. Such concerns must also be carefully addressed; once a device provides data and receives commands more openly, the *thing* architecture must "pick up the slack" and ensures these APIs are not misused or abused. Even when the API is being used properly, health devices must consider identity, or *who* is using the API; a primary user may be able to see readings from a device through its API, but these readings should not be available freely.

3.3.2 User Interface

Many devices follow the trend of the temperature sensor described in the previous section; the hardware is physically quite minimal, with a limited number of small buttons or information displays. This subjects them to a variety of usability and accessibility concerns, even when considering them along with their companion apps. While simple functionality can be exposed to other devices or controllers through standardized APIs, this may not work as well for more unique or advanced features. Instead, a common set of user interface-focused API endpoints could help alleviate these issues, allowing a user's preferred or specialized application/device to manage the display logic and interface with more complex behaviors.

Accounting for a wide range of accessibility requirements is a difficult task; however, democratization has the potential to simplify this issue. Consider an API that manages rendering a desired accessible interface. Such an API could take in accessibility elements as parameters and output a device-agnostic markup to be managed by the mobile app, specialized device, or other means of output. This splits the requirement more evenly amongst the thing, the output device, and the user. Markup languages such as UsiXML [54] attempt this, allowing an interface specification to be used across multiple UI contexts (such as graphical, auditory, etc.)

XIS-mobile [55] is another UI language to describe mobile device interfaces in specific. In simple cases, a UI may even just reflect the thing's underlying API.

3.3.3 CHANNELING

Note that many of these devices mentioned above likely offer limited support for data sharing with other ecosystems (for example, information may be exported as a file from the cloud, such as CSV or XML). Within health IoT, channeling describes the right and ability of the user to share data from a device to the greater healthcare ecosystem. Many health *things* collect measurements and store data either locally on the device, or in a cloud service managed by the device vendor. In either case, the data is likely stored mainly for use within the ecosystem, with little provision for external transfer. The user, however, may want to share this information with their healthcare provider (for example, as external notes or as an addition to their electronic health record), emergency services (such as when the data represents a dangerous condition requiring immediate attention or intervention), or even themselves (in personal storage or a private cloud). Because data on many of these health devices can be sensitive and must be protected, access cannot be given out freely; instead, an effective health architecture must assist in this obligation and consider the data access rights before sharing it with various third parties.

For example, a user's smart fitness band may be a part of the Apple Health ecosystem [6]. This platform attempts to democratize the storing and channeling of health data, allowing users to collect data from supported health IoT devices and permit specific parties access to this data. On the device side, this is achieved through HealthKit [56], a set of common APIs used by vendors to share data and events from their supported apps. To channel the data generated by these supported apps, the platform uses the Fast Healthcare Interoperability Resources (FHIR) [57] standard to integrate with supported healthcare providers' electronic records. While this is a step towards health IoT democratization, the user is still tied to an ecosystem at a larger scope: the data is not truly cross-platform. A user

wanting to switch to Android and Google Fit [7], for example, would not be able to transfer their data directly through first-party services.

3.3.4 Requirement Specification

Section 3.3 introduced a variety of user-facing democratization features that aim to improve the compatibility and functionality between *thing* devices and the interfaces through which they may be interacted with. These features are summarized into the following requirement specifications:

- *Shared APIs:* A health *thing* should expose its core functionality through a set of open, well-defined API endpoints.
- *User Interface:* A health *thing* should consider the possibility for alternative or new user interfaces, both physically and in software.
- *Channeling:* A health *thing* should ensure its data is available to and shareable by its owner or end users.

As these requirements focus more on the *thing's* interaction with the end user, they are likely be significant considerations in the design of specific health devices and applications. Still, these requirements contain a variety of aspects that guide the design of the generalized *thing* architecture described in chapter 4. For example, the IoT-DDL specification (subsection 4.2.1) revolves around the Shared APIs requirement, and the DIY Health IoT Apps project (section 4.3) works towards the User Interface requirement. The concept of Democratization remains a consideration throughout the Atlas Health Architecture, and guides the experimental design and validation described in section 6.3.

3.4 Requirement II: Device IoTility

As discussed previously, a device's ability to effectively participate as an IoT *thing* is influenced largely by how it is prepared to interact with users, other devices, and the

smart space as a whole. This ability to enable meaningful and safe interactions is referred to as a *thing's* IoTility within this thesis. The concept of IoTility covers a number of related requirements (individually described in this section) that center around utilizing a health device's connectivity features: taking full advantage of the information and functionality available within its surroundings to augment and improve the services the device may offer.

These capabilities are very relevant to many digital health scenarios, where devices monitor or interact with the user directly and share closely related goals involving management of the owner's health. For example, consider a patient that has an elevated risk for diabetes and related complications. Following consultation with their physician, suggestions are made for the patient to make lifestyle adjustments in an effort to reverse this risk. The patient begins an exercise regimen, purchasing a smart body-weight scale, fitness tracker watch, and pulse oximeter, and downloading their associated smart phone applications. Each of these devices take related measurements and aim to assist the user in reaching and maintaining a healthier lifestyle. A short time later, the patient becomes more invested in their health and potential risks, and purchases a blood glucose meter and blood pressure cuff to more accurately monitor for symptoms of prediabetes. As the patient's health and treatment requirements change, they may continue to acquire more devices to provide further measurements and features supporting their health goals.

While these devices—many based on traditional healthcare instruments—each fulfil a specific purpose, considering them independently means the user must manage and make sense of their information to see their bigger health picture. Introducing a high-IoTility health architecture, as described above, increases the potential of these devices to better meet the needs of the user and their health goals. Allowing these *thing* devices to work together and share data can open up many application possibilities on the principle that "the whole is greater than the sum of its parts."

However, before these devices can interact meaningfully, it must meet some basic requirements to communicate with and understand its smart space. To enable

interaction with an entire smart space, a device must communicate despite language or protocol differences, such as through a "translator" *thing* [43]. To understand various opportunities for interaction, a device must learn about the available APIs, services, and semantics of other things (as well as share its own) through information exchange. To act on these opportunities, a device must also be able to infer various cooperative and competitive relationships with the other things in the smart space [44]. Together, these features lay the groundwork for the formation of *meaningful interactions* and IoTility in a health IoT smart space.

Beyond direct data sharing, the interactions and events generated by a health device can be used to infer pieces of user context. This refers to information that helps a device describe "when" it should be used; critical when considering health devices that require use in specific intervals throughout the day or after certain events. For example, a glucose monitor must be used two hours after eating a meal to achieve the most accurate readings. Traditionally, the user may need to remember or manually set an alarm; however, if the smart space is aware of when the user eats, the timing and reminders could be managed by the glucose monitor device itself. This kind of context usually cannot be deduced from a single device; instead, it must be "built up" over time using a combination of events from different devices and measurements.

Continuing on the previous example, it is unlikely that the glucose monitor can detect when the user eats on its own. However, the user may enter their meal information into a diet-tracking app, or use a smartwatch app that automatically detects eating movements [66], which can then share this contextual information for use by the glucose monitor. Measurements from multiple devices can also allow the users or their caregivers see a bigger health picture; while a single abnormal data point may not be a concern, unexpected readings across multiple devices might point to a larger issue. Achieving these goals through meaningful interactions with fellow smart space *things* is especially important when considering the variety of constrained or low-power health IoT devices that feature limited independent capabilities.

3.4.1 Relationships Between *Things*

Communication between *things* is a substantial part of an IoT ecosystem. The advantages brought by democratization (as described in section 3.3) are less impactful when considering only interactions between a *thing* and the user. Once *things* are able to "speak" some level of a common language, they can interact not only with the user and edge, but also with each other. This *thing*-to-*thing* interaction is another critical part of a functional health IoT system: a *thing* with the potential to cooperate and directly utilize the capabilities of the smart space increases the capacity for meaningful interactions to occur. This is especially true in an environment with a wide variety of health sensors and devices. Even if these devices share a common interface, they still must be individually considered and programmed for, which quickly becomes unreasonable when considering a large number of devices and the various potential synergies between them.

Relationships, or logical links between functionalities offered by two or more *things*, allow for the creation of implicit and opportune interactions between smart space devices. A relationship may allow a thing to become a conditional element within a logic matrix, its output used as an input or to control another thing, as in IFTTT [29] and TTEO [31]. A *thing's* ability to form these types of relationships (especially when they can be formed as suggested by the *thing,* rather than explicit user intervention, such as the Social IoT [32]), can help users focus on the high-level functionality of their smart space, leaving the low-level details to be taken care of by the architecture itself. These kinds of relationships are especially useful in a personal health environment, where sensors may only record a part of a larger metric (such as how body temperature, blood pressure, etc. make up a patient's general vital signs), or may record the same measurement in conjunction with other sensors (such as reading pulse in different places on the body). Such grouping of information can be handled or specified between *things* through relationships, reducing the need for explicit programming and simplifying the use of the smart space at the edge,

especially for personal health scenarios where the user may not want or know how to effectively manage their array of smart devices.

Consider a patient with a blood pressure measuring device and pulse oximeter. Both of these devices are capable of recording, amongst their other measurements, the patient's pulse. In the traditional case, the user would be presented with two pulse readings, and would likely choose to use one over the other, possibly hiding or removing the second measurement. Imagine instead that these devices look to form relationships with other devices (whose specific form may not be known) that provide a pulse reading, allowing them to combine readings or keep each other in check. Such a situation improves usability and allows for a form of reliability across devices taking the same measurement (where services only need to be aware of the measurement itself, not the source device). Handling these relationship-based behaviors within the architecture itself can help supplement and simplify the creation of meaningful interactions without relying on specific knowledge from the *thing* app or software.

3.4.2 Thing-Like Mobile Apps

In many IoT scenarios, the mobile app plays a prominent role in the interaction. The number of these apps targeting digital health is significant: over 300,000 as of 2017 [67, 68], and still growing. In addition to self-contained health apps, many health IoT devices require a mobile device to host their companion app. However, despite their close interaction with *things* and the smart space, these mobile apps are not utilized fully: this thesis argues that they can be further integrated as *things* themselves. Considering mobile applications as "software *things*" (as opposed to hardware *things*, such as sensors, actuators, or other personal health devices) can open up new potential for meaningful interactions within a smart space. In most situations, the mobile app either acts as a controller for other smart *things*, such as requesting data or sending commands, or manages information for the user, such as the aforementioned diet app. In terms of a smart *thing*, this is only half of the picture. Other *things* in a smart space cannot consider the app for any interactions beyond

what it is programmed to do. Even if a health app's information could be used by another *thing* (especially one un-related to the purpose of the app), the app most likely lacks a way to channel data to that *thing*. The app is missing a concrete API like those of other physical *things*, simply because most mobile apps are not designed to work this way.

For example, consider a patient with a dieting app and a smart fitness band. The user eats a meal, enters the information into their app, and receives an updated exercise plan. The user then must use another mobile app or other method to update their fitness band's configuration, to reflect the exercise parameters they were given. Even though all of the information is available implicitly within the smart space, the user must manually "transfer" parameters between the apps and the *things,* because the developer of the app did not necessarily consider all potential interactions. If the diet app had a *thing*-like API, the interaction could have progressed differently: after the user enters their information into the app, the fitness band sees that the exercise plan has changed, updating its parameters automatically. In this case, the *thing* becomes the driver of the interaction, where it asks the mobile app for information through its *thing*-like API. Although the app was not developed with the fitness band in mind, it was able to provide its information to the smart space and enable this transfer of parameters.

Compared to a normal hardware *thing*, a mobile app is likely much more capable in terms of features and the ability for the user to interact. *Things* in the smart space can potentially utilize a wide variety of sensors (for example, accelerometer and GPS) or engage the user through a touchscreen interface. This is especially interesting for lightweight hardware *things* with minimal physical interfaces, like the body temperature sensor example mentioned in section 3.3. Such functionality makes higher-level interactions available to *things* without the need for physical interfaces or a specially programmed app (thereby reducing the need for the "silos" created by many existing companion apps). Used with the concepts from the previous section, mobile apps could even form *thing*-like relationships with other

devices or even other apps, allowing users to “combine” app functionalities, or easily create new app-like behaviors with the exact functionality they desire.

3.4.3 Proxy Interfaces

Many health IoT devices offer relatively minimal physical interfaces, deferring many of their interactions to a companion mobile app. While this is usually convenient or even preferred, it is also important to consider times when a user does not have their mobile device on their person; such a situation may occur while charging the device, or with an elderly patient that does not use their device frequently. Even without a mobile device nearby, other smart things may be able to convey information to or accept input from the user. With the appropriate context and shared information, any device could act as a proxy for interactions with other things, either through their APIs, hardware user interface elements, or both.

In this case, a proxy interface describes a device used as a host for another thing’s functionality. In fact, a mobile companion app can be considered a proxy interface for its respective health IoT device. Some devices may use a proxy interface in addition to a basic physical interface, while others may use this external interface exclusively (such as in simple sensor devices without displays). In these cases, loosening the relationship between a device and its mobile app controller could help enable new simplified interactions or allow information to be shown with greater precision, relevance, or availability within the smart space.

The efficacy of these proxy interfaces hinges on the democratized capabilities of their device APIs, as described in section 3.3. In addition to offering its own services, a device may want to discover the capabilities of other things in the smart space, to search for potential proxies or to offer its features to less capable devices. Similarly, a proxy interface may even exist solely between two devices; for example, a Z-wave device’s services may be proxied as REST endpoints by a separate device that also has WiFi hardware. An architecture must support this ability to share capabilities and form such relationships to fully enable this potential for meaningful interactions.

3.4.4 Notifications and Reminders

Notifications, including reminders, device and health information, and other forms of non-transactional incentives, are another important set of features that can support and empower the user. Many health IoT devices, such as fitness bands and other sensors, are meant to be used passively: most interaction is done in retrospect to view collected data over time. In these cases, different forms of notifications are especially useful to convey data in situations involving abnormal readings, contextual data, and other time-sensitive information. Under normal use, the user might not receive such information until far after the fact; a notification allows the device to capture the user's attention at the time of occurrence. Such notifications are already a well-understood interaction in the mobile application space, making them very useful for communicating important information from things.

In addition to traditional *thing*-to-smartphone notifications, alerts may be transmittable between health devices. Even if two devices do not interact through their APIs directly, a generalized event may be broadcast by one, allowing the other device to react despite not explicitly supporting the source device. This is powerful when considering proxy interfaces and the scenarios described above, where a user away from their mobile device may miss an important notification. A capable device, such as a fitness band with LEDs or a speaker, could pick up this notification and gain the user's attention where not normally possible without a smartphone.
Reminders, a subclass of notifications, extend these concepts and focus on empowering the user through specifically timed notices. These types of notifications are especially important within a health IoT ecosystem, where they may help ensure a user continues to use a device as required, such as maintaining a schedule or taking measurements at specific times. For example, a typical blood glucose meter requires the user to take a measurement two hours after eating: the device could schedule a set of reminders on the user's phone, describing this requirement and triggering a notification at the appropriate time. The ability to meaningfully react is especially important for this type of notification: the glucometer, for example, may not be

programmed with time-tracking functionality, and instead requests another device (the user's smart phone) to set up an alarm with its desired parameters.

3.4.5 INCENTIVES

Compliance and adherence when using personal health devices faces issues resembling problems with medicine compliance [69]; that is, taking medicine on time, consistently, and taking the proper dose). Patient empowerment is an effective strategy to achieve compliance or at least convergence (where attempts to influence or change behavior begin to have an effect), especially in patients with long-term conditions [70]. However, a persistent problem in this form of empowerment is the lack of a roles model: "who does what and when?" The concept of device IoTility argues that the connected health device itself can contribute and play a role in patient compliance and empowerment. Evidence would be needed to establish that such a device contribution could make a difference, however, this would only be possible after an implementation of the requirement feature is included in at least some health devices.

The three key elements of empowerment and successfully engagement are: recall (reminders), reward (incentives), and maintaining capacity and context (encouraging actions that are possible and doable by the user) [71] [72]. Measuring and operating within user capacities is difficult, and although it may be approximated under the context of how empowerment attempts are delivered, this element is not addressed in this thesis. Recall is well addressed under the IoTiity requirement, as shown in subsection 3.4.4. The last element, incentives, also remains important even through it is a broad concept and difficult to architect; implementations would be likely to use external elements and assets such as credit, passes, discounts, or anything of value to the end user that could be provided by healthcare systems or providers. Given this generality, the IoTility requirement for incentivization would likely follow a cooperative process that places the device in the pathway of managing these incentives. For example, a device may report usage statistics periodically to a democratized app or server (such as those described in section 3.3). This could

include data on timing and accuracy (using concepts from section 3.5), such as when the device was used and if it was used properly over a period of time. The receiving end may accumulate this information and "score" it to convert into incentives; this is where the unknown external possibilities would fall into place. Interfacing with these unknown possibilities could occur through smart contracts [73] (connected to by external organizations or businesses) or a form of digital wallet (such as Apple Wallet or Google Pay).

3.4.6 Requirement Specification

Section 3.4 introduced a selection of system and device-focused IoTility features that aim to facilitate local interaction and cooperation between *things* in a smart space. These features focus on the vendor and developer stakeholders, describing behavior that can enrich IoT interactions without direct involvement or contribution from the end user. These features are summarized into the following requirement specifications:

- *Relationships Between Things:* A health *thing* should be prepared to interact with other devices through some shared protocol.
- *Thing-Like Mobile Apps*: Mobile applications and software should be able to interact with a smart space in the same manner as hardware *things*.
- *Proxy Interfaces:* A health *thing* should be able to act as an intermediate for other devices or alternative communication methods.
- *Notifications and Reminders:* A health *thing* should be able to utilize notifications and reminders in the same manner as a mobile app.
- *Incentives:* A health *thing* should consider the role of empowerment and engagement in the design of their behavior and features.

As described above, these requirements represent a variety of features that can be provided by the *thing* architecture and utilized in a variety of health applications. As a result, some of these features are implemented directly in chapters 4 and 5 (such

as Mobile Apps As Things in section 4.4 and Proxy Interfaces in subsection 5.4.3). Other architectural features provide more flexible support for a variety of requirements, such as Tweets (subsection 4.2.2) and the relationships provided by the Atlas Thing Architecture (subsection 4.1.1). Together, these requirements also represent the bulk of the experiments and validation in this thesis, resulting in evaluations described in sections 6.1 and 6.2, as well as a portion of section 6.4.

3.5 Requirement III: Safe Use

Safety and accuracy are both primary concerns in healthcare environments in many regards, from correctly taking a measurement to administering a proper dosage. While the influx of connected medical devices has helped make these actions easier and more accessible than ever before, safe use remains a prominent interest. Especially when considering the number intended for use by inexperienced end users, health IoT devices must not only become easier to operate correctly, but also harder to use in a wrong or unsafe manner. For a health IoT architecture, safe use refers to a thing's ability to correctly and reliably interact with the user using both its inputs and outputs.

For sensor devices, safety manifests through the capacity for detecting proper use: ensuring the user is taking a measurement the right way, at the correct time, etc. To achieve this, an architecture must have some built-in capacity for error detection on its inputs; depending on the device's function, this may be possible solely through software, or may require the assistance of additional hardware. For example, a smart watch EKG device may use software signal processing to determine the quality and validity of its readings. On the other hand, an automated blood pressure device may require state from additional hardware elements such as accelerometers to determine proper cuff placement. Once erroneous input is detected, an architecture must be able to use this information to guide and correct the user. While data can be marked as valid or invalid for further processing within the device or other things, the user is likely to continue providing this bad data

without feedback on adjusting their interactions. Depending on the device, this may involve conveying a binary value, an accuracy (analog) rating, or more traditional feedback such as additional instructions. For example, the Kardia EKG device [58] displays a real-time "signal-strength" graphic to guide the position of the user's fingers, while the A&D Medical wrist blood pressure monitor [59] controls LEDs signaling the user to move their wrist lower or higher.

For some devices, this feedback may be provided in real-time, allowing the user to adjust their behavior and prevent an invalid reading from occurring in the first place. Even in devices (especially those with "dumb" components, such as the test strips of a user's glucometer) where this is not possible, an invalid reading may be detectable such that it is discarded, and the user presented with additional guidance or coaching to repeat the process correctly. This simple form of input safety becomes valuable when considering users who own many devices or are less familiar with a device (which may even be prescribed by a physician rather than personally chosen), especially compared to more traditional assistance such as physical instructions [60].

Compared to sensor devices, safety for actuator devices mainly involves detecting behavior or commands that could harm the user or physically damage the device itself. While input safety can process an invalid reading and discard it after the fact, output safety must always handle potential errors before the action actually occurs. Health IoT devices cannot stop all improper use, but an architecture can consider and manage how a device is being used through its software APIs; for example, an impatient user pressing a button five times does not necessarily correlate to five sequential or simultaneous requests to the *thing's* functionality. While actuator devices are less common amongst health IoT devices, they are often "high-risk," higher-tier (medical) devices; for example, output safety would be critical in closed-loop insulin pumps (which can provide more personalized treatment regimens [61] while still boasting an impressive safety profile [62, 63]) or other dosing devices. Similar safety features are still important in lower-risk actuators, such as a blood pressure monitor preventing over-tightening or damage to the air pump.

Due to their inherent risk, many of these actuator devices are likely to have similar safety features in place already; however, further integration with health IoT ecosystems will likely bring about new forms of interaction that require additional precautions and considerations in regard to the safety of these devices.

An architecture may validate these requests through constraints [64], or additional validation on the conditions of an action. In actuator devices, this includes limitations on aspects such as the frequency of invocation, the range of a parameter, or the times when an invocation is allowed. For sensor devices, constraints focus more on the results of the process, considering output value ranges or the states of components within the device. In either case, a triggered constraint may then be used to provide additional feedback to the user, as described above. Such enforcements help ensure a device stays in a functioning and expected state, regardless of how the user interacts with it. This type of safety also extends to "fail-safe" modes [65] for actuator devices, where a malfunction or loss of power, such as a health IoT wearable running out of battery, would not leave the device in a state that could cause harm. How an architecture offers and manages these safety features is critical when considering the effective use of various health IoT devices.

3.6 Requirement IV: User Identity and Privacy

User identity is another critical component of a health IoT ecosystem; many personal health devices will need some concept of an active or owning user. Most obvious are the larger or home-based devices, such as blood pressure monitors and body weight scales, that are shared between members of a family or household. If the device intends to share its readings with the smart space or a user's app, it must first know how to associate this information with the correct user. Such a device may rely on the user or the connectivity of other devices in the smart space to determine this. For example, a glucometer may prompt the user to select their profile, or the companion app (specific to each user's mobile device) may share this information during use automatically.

When considering the connectivity features described in section 6, identity becomes a prominent consideration even in single-user wearable devices, since such potential interactions should usually be limited to devices owned by the same user. For example, the glucometer scenario above should not set reminders based on a family member's dieting app. In this case, while a device may not track multiple user profiles, it must still associate identity with broadcast measurements, or remember which devices share ownership and are clear for interaction. These concepts of user data and ownership are therefore critical within an architecture, as they directly affect how a device can discover and interact with its smart space.

Privacy is also a primary concern when considering user identity and how health IoT devices share information: the data and functionality offered by many devices should not necessarily be shared freely with an entire smart space. In addition to ownership, a health IoT thing must also consider who else (if anyone) a given piece of data should or could be made available to. This is especially true when considering devices that support democratized channeling, as described in subsection 3.3.3. Data made available to third parties or healthcare providers may also become protected health information, increasing security requirements and complicating matters for a thing that shares its information within a smart space. In this case, a strong and secure user identity is essential to the health IoT device.

These concepts play a role when integrating incentivization features as well, such as those described above. These features and their associated measurements will likely require some level of verification (in terms of the user and device identity of the data) that a platform can trust when calculating how to administer a reward. For example, a user receiving incentives for consistent use of a glucometer should not be rewarded when a family member wants to take a reading for themselves. In addition to the forms of detection described above, a health IoT device may integrate some form user authentication; this could involve a biometric challenge or defer to the hardware present in the user's mobile device, such as fingerprint scanning and facial recognition. Such authentication features also bring into consideration how health IoT things are provisioned in a new smart space. For many

single- and multi-user health devices, identity is a main focus during the setup procedure. A health device may require a variety of identifying features to function; these can be specified manually, or even sourced automatically from parameters in the smart space [20]. The way an architecture handles this procedure is critical in determining how a device may be used; some devices may require a configured identity before use, while others function even in an un-provisioned state (this scenario is also discussed in section 3.3).

3.7 SUMMARY

This chapter introduced a set of requirements intended to meet the specialized needs of health IoT. First, a high-level methodology was presented, describing the process of identifying, implementing, and evaluating a new requirement. This methodology also highlights some of the considerations taken during the selection of the presented requirements. One such consideration is the difference between the "system" and the "ecosystem;" while the main concern of the IoT architecture involves the hardware it controls, the device's interaction with other devices, applications, and services also plays a large part in influencing the architecture's design and features. The methodology also considers the role of the end user in relation to the device architecture: while the needs of the user set the goals of the architecture overall, they may not map to individual system features and requirements. Instead, requirements may also stem from stakeholders such as vendors and developers, who create applications using the architecture.

This chapter also introduced the concept of IoTility, or the ability to increase the collective utility of a group of smart *things*. This concept relates closely to the potential functionality of many health IoT *things*, and represents one major grouping of the identified requirements (alongside Democratization, Safe Use, and User Identity and Privacy). The chapter then considers these requirements across the full spectrum of health IoT, considering their use in *wellness, personal health,* and *medical* scenarios. As each of these "classifications" are likely to present different

needs and use cases, the requirements are discussed mainly in the context of *personal health*: devices that may contain more advanced health features but are meant to be used by individuals outside of a hospital setting.

The chapter then investigated each requirement grouping in more detail, focusing especially on Democratization and Device IoTility. These two requirements are broken down further into individual features and concepts that may be integrated into a health IoT architecture. Democratization focuses on how users may control or decide how to use the *thing*, arguing for features such as common user-facing programming APIs, additional focus on user interface capabilities (especially when considering accessibility), and data transfer and ownership between users and their healthcare providers. On the other hand, Device IoTility focuses on how a device may enrich the user through the combined capabilities of other *things*, without requiring explicit command or intervention. This includes concepts such as creating *thing*-like mobile applications, proxying interactions between dissimilar *thing* devices, and integrating incentivization features and interactions.

These requirements are used to guide the design and implementation of the architecture described in chapters 4 and 5, especially those related to the IoTility requirement. Still, while these requirements introduce a variety of powerful features prepared to meet the needs of health IoT, the chapter notes the possibility of additional or modified requirements as the architecture evolves. Potential limitations of the selected features, as well as directions leading to new or modified requirements, are discussed further in chapter 7.

CHAPTER 4: A HIGH IOTILITY HEALTH ARCHITECTURE

Fully supporting the requirements described in chapter 3 calls for careful consideration of their impact and position within the lower levels of an IoT device; that is, the *thing* architecture. While it is possible to address some of these needs within an individual IoT application, a focused architectural implementation can position the *thing* to more easily and consistently offer these important functionalities across a variety of devices and scenarios; points that are critical when considering the fragmented and evolving nature of the personal health ecosystem. A health IoT application may then utilize or build on top of the functionalities provided by this groundwork to better meet the specific needs of their end users. To this end, the architecture presented in this thesis uses such requirements to guide the implementation of core features that can facilitate new meaningful interactions with users and other *things* in a health IoT ecosystem.

While all of the previously defined requirements play a role in forming these features, those related to the concept of IoTility (detailed in section 3.2) are especially significant in the context of an IoT architecture: they influence how a *thing* is prepared to interact with the rest of the smart space. Considered on its own, a health device is mainly concerned with performing its physical functions (sensing or actuating); here, an architecture may be able to provide hardware abstractions or other utilities. Expanding to a traditional ecosystem silo, where the device communicates through a mobile application or the cloud, an architecture may become more involved, managing safe use, networking, and security concerns. Beyond this, much of a *thing* architecture's potential stems from how a device can be programmed (by the vendor, developer, and even end user) to support new "high IoTility" interactions that specifically target the specialized needs of health IoT.

This chapter presents an overview of the developed *thing* architecture and its prominent components, each of which aims to provide new features that satisfy one or more of the requirements identified in chapter 3—especially those related to the

concept of IoTility. Section 4.1 introduces the Atlas architecture, and describes the modifications made to support new health IoT requirements. The remaining sections, 4.2 through 4.4, focus on individual components that work within or alongside the architecture to further enable the meaningful interactions described above.

4.1 The Atlas Health IoT Architecture

The Atlas Health IoT Architecture described in this thesis builds off the structure of the Atlas Thing Architecture [41, 40]. This earlier architecture aims to better enable *things* to self-discover their characteristics and capabilities, which can then be shared within a smart space with other things. Such "social" *thing-to-thing* interactions are a core focus, versus an architecture that communicates through a single point such as a cloud service. The new health architecture modifies these components to better support IoTility features and other health IoT requirements, while also introducing new capabilities that work towards this same goal. The following subsections provide an overview of the original Atlas Thing Architecture, as well as a detailed description of the new Atlas Health IoT Architecture.

4.1.1 The Atlas Thing Architecture

The Atlas Thing Architecture is a set of software operating layers that provide functionalities a *thing* requires to engage and interact meaningfully within a smart space, first introduced by Khaled [39]. The Atlas platform (the upper block of figure 4-1) itself exists on top of a set of base *thing* operating system services, abstracted by a host interface layer that provides portability and interoperability. These layers manage the interactions between the Atlas platform and device features such as networking, memory management, I/O interfaces, etc. Building on top of this, the architecture consists of several elements focusing on different issues in IoT, each of which are represented through the individual blocks within the Atlas platform.

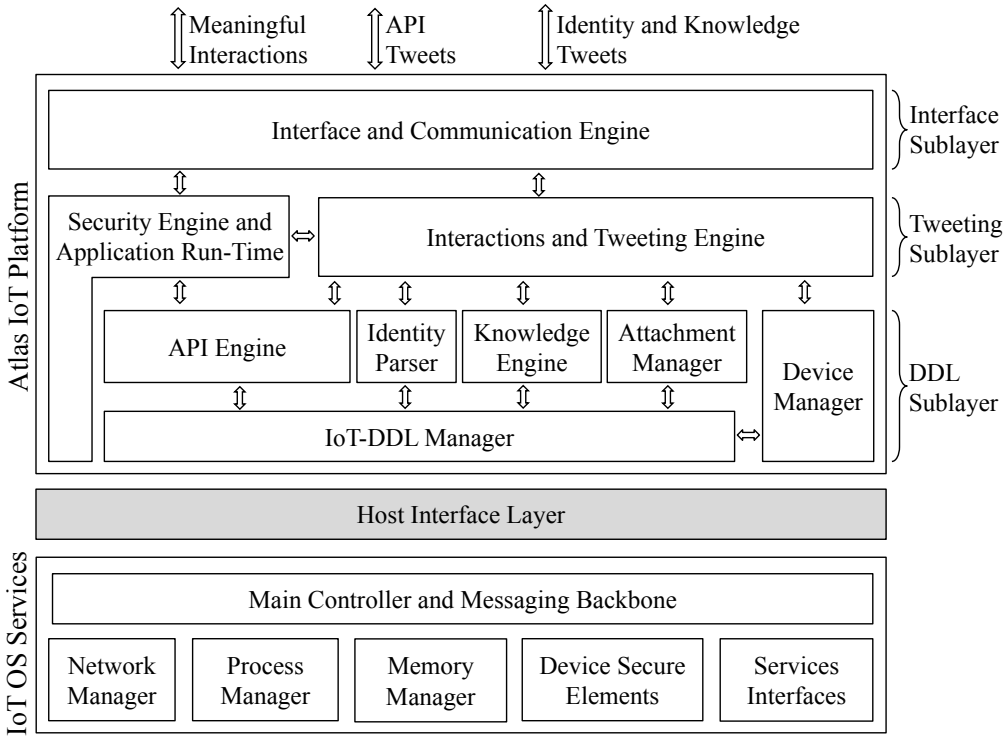

Figure 3-1: Overview of the Atlas Thing Architecture [39].

At the lowest level, the *API Engine* manages a device's service behaviors and endpoints. The structure of these services is derived from the *IoT-DDL Manager,* which loads *thing* information from a structured device description (described in section 4.2.1). The manager also provides metadata information that can be shared through messages (described in section 4.2.2) formatted by the *Interactions and Tweeting Engine* and sent to other devices in the smart space. Messages received from other *things* can then facilitate new interactions and relationships [44] through the *Knowledge Engine*. Additionally, the *Security Engine* manages authorization and encryption of active *thing* interactions, and the *Interface and Communication Engine* manages converting any messages to various protocols such as REST and MQTT [43].

Together, these components create a powerful *thing* architecture targeting the traditional personal IoT domain. It is important to note that the Atlas Thing Architecture mainly focuses on more powerful *thing* devices, including boards that can run full-featured operating systems (such as the Raspberry Pi [74]) and microcontrollers with more advanced networking and data storage features. On a

more constrained device, some features may not be feasible: for example, a *thing* lacking a WiFi radio cannot share metadata information in the same manner as a more capable *thing*, and a device without flexible storage may be unable to load its device description at runtime. These constraints are an important consideration when adapting to the personal health domain, where devices represent a large variety of capabilities and may vary in their ability to support a specific health IoT requirement.

4.1.2 Adaptations for Health IoT

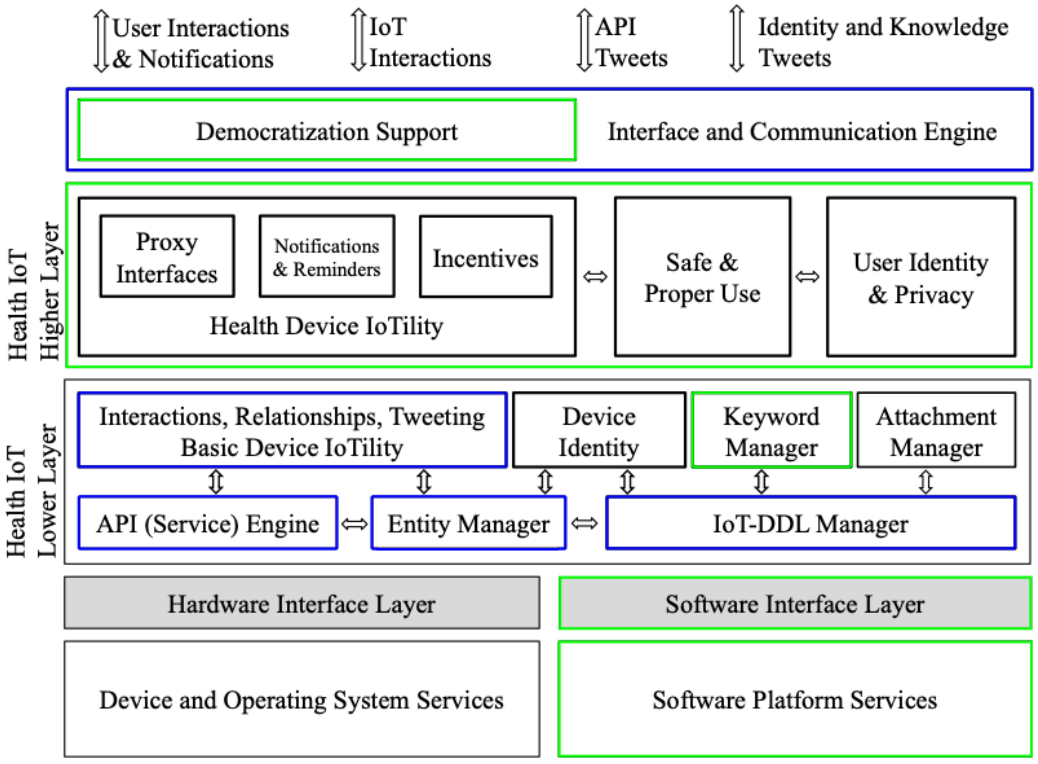

Figure 4-2: Block diagram of the Atlas Health IoT Architecture.

The Atlas Health Architecture utilizes and augments the features of the Atlas Thing Architecture to better target the goals and requirements of health IoT, as shown in figure 4-2 (where blue outlines represent modified components and green outlines represent new ones). In the *Lower Layer*, the Atlas Platform components are built upon to provide the base for the IoTility features described in chapter 3. For example, the *API (Service) Engine* and *Entity Manager* are adapted to a microservices

architecture (expanded on in chapter 5) that supports a variety of API and IoT interactions. Some new components are also introduced, such as the *Keyword Manager* to handle information collected from tweets (described in section 4.2.2) and mobile apps acting as *things* (described in section 4.4). Above these components, the *Higher Layer* utilizes these core platform features to extend and adjust interactions as they are passed to and from the Communication Engine above. In this layer, the components provide self-contained IoTility capabilities that can be utilized directly in specialized Health IoT interactions. As shown in figure 4-2, these components are named after the architectural requirements they target:

- The *Safe & Proper Use* component manages a set of pre- and post-conditions for each API endpoint. For example, a received API call (an IoT interaction) may be prevented from reaching the API Engine if a parameter value is outside of its expected range. Such "constraints" may also depend on the physical characteristics of the device or environment, the state of other *things*, and the time and frequency of interaction. This component is described in further detail within chapter 5.
- The *User Identity & Privacy* component manages any user identifiers and, along with information from the Device Identity, packages this data with service outputs as needed. The component also provides basic security features, such as enabling the encryption of inter-thing communications. Like *Safe Use*, this component works closely with the API Engine, but can still be managed independently.
- The *Health Device IoTility* component contains a variety of features that support the IoTility concepts described in chapter 3, including managing notification-style communications and exposing endpoints and capabilities for incentives-based reporting. The component also allows a device to proxy service calls to other devices with different capabilities, allowing a greater variety of devices to participate in the interactions enabled by the architecture. The details of these IoTility features are expanded on alongside the discussion of their implementation in chapter 5.

Above the Health IoT layer exists the *Interface and Communication Engine*, which contains the democratization component of the requirements. This layer manages the communications received by the health *thing*, including IoT interactions (such as service calls) and user interactions (such as notifications). The democratization component manages representations of the API endpoints from the API Engine, allowing standardized services to interact with the device, and enabling effects such as an authorized, user-approved "redirect" of a native health device companion app to a separate democratized one. Together, these components enable many of the requirements presented in chapter 3—creating a foundation for a variety of personal health IoT applications.

The separation between the Lower and Higher Layers is also an important feature of the architecture. While most of the Lower Layer components are closely integrated and related, the Higher Layer features offer greater “standalone” capabilities; they may still be useful even in scenarios where the complete Atlas Platform components are not supported. Imagine a scenario similar to the one described in section 4.1.1, where a minimal *thing* platform, such as a low-power BLE device, cannot offer the IoTility features integrated within the Lower Layer (such as IoT-DDL loading or tweeting over WiFi). Even without this support, Higher Layer features such as Safe & Proper Use may still be important requirements for the *thing*. Defining this separation ensures the utility of a requirement can be maximized across a variety of health IoT device classes.

Another important feature of the new health architecture components is represented by the addition of the *Software Platform Support* block, positioned next to the *Device Operating System Services* in figure 4-2. In addition to traditional *thing* devices, this thesis highlights the “software *thing*,” such as a mobile application, as an important part of a health IoT ecosystem. In such a scenario, the *thing* is no longer represented only by physical hardware, but possibly by an ephemeral application that exists as a part of a larger system as well; the architecture interfaces less with low-level features and acts more as a software library providing IoT functions to the

application. Like the hardware device services, these features are abstracted by an interface layer sitting below the core architecture. These services are significant for such software *things,* as their needs and functionality likely differ from a similar hardware device: for example, a mobile app *thing* might manage interactions and offer services defined by the state of the application rather than a manually created device description (IoT-DDL). Here, while the higher-level health IoT requirements are similar, the Atlas platform features are used in a different capacity. Similar situations relating to mobile apps are explored further in sections 4.3 and 4.4.

4.2 Expanding IoTility in the Health IoT Lower Layer

Present-day IoT platforms and architectures [75, 76] often revolve around communication through a central point (such as cloud platforms or an edge) where users can control *things* and access data. This type of architecture is highly effective for cloud-based application development; however, it is often vendor-specific and ignores the potential for direct communication between *things*. These silo-constrained connections limit the capabilities of the smart space, preventing *things* of different type or vendor from fully engaging. To truly realize the potential of an IoT ecosystem, *things* should be able to seamlessly integrate with clouds, edges, and devices across a variety of vendors and developers: features that are a primary focus of the Atlas Architecture. The following subsections expand on two components that are central to this potential, the *IoT-DDL Manager* and the *Interactions and Tweeting Engine*, and present aspects that have been developed further to better enable core IoTility features and increase democratization and interaction between health IoT *things* within the Atlas Health Architecture.

4.2.1 The Internet of Things Device Description Language

While democratized communication is important, simply connecting devices that "speak the same language" is not enough to fully realize the potential of the IoT. Rather, this also depends on how a *thing's* services, resources, and capabilities are

described and made known to the smart space, and how a *thing* can utilize that information to support promising IoT scenarios. The main constraint in achieving this centers on the wide heterogeneity of communication technologies, environments, generated data, and measurement capabilities that may be found on a device. Such variability introduces a significant challenge in integrating these *things* with a surrounding smart space and IoT ecosystem.

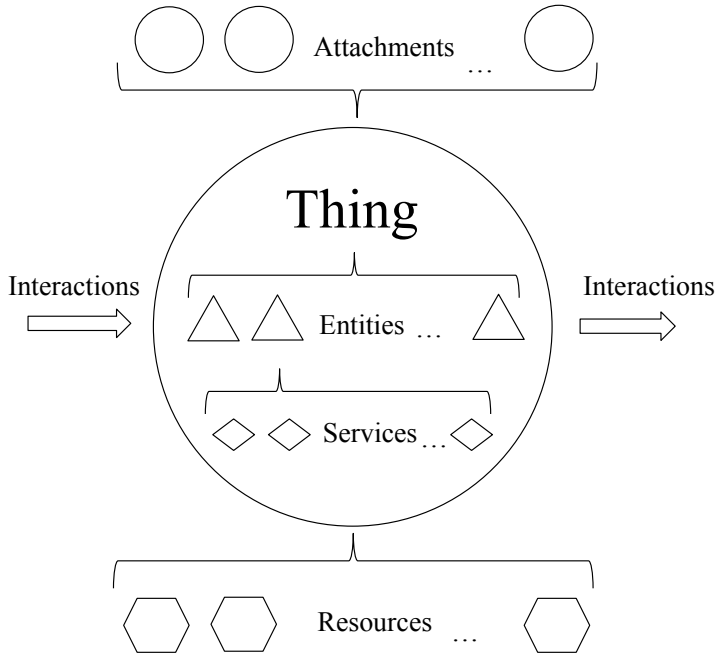

Figure 4-3: Generalized thing structure and major IoT-DDL components.

The Internet of Things Device Description Language (IoT-DDL) defines a human- and machine-readable schema to structure and address some of these democratization requirements. This schema builds on top of the previous Atlas DDL specifications [38, 40], shifting focus from discovering hardware services directly to enabling ad-hoc connections between the various capabilities within a set of *thing* devices. To achieve this, a structure must be identified that represents not only the potential characteristics of the *thing*, but also the variety of forms and purposes it may take. For example, a *thing* may be hardware-based (such as a sensor or actuator), software-based (such as a mobile app), or even a hybrid that offers a variety of

services to the smart space. From another perspective, a *thing* may be a single low-power sensor, a powerful interactive device, or a collection of different *thing-like* entities. These types and roles influence the kinds of meaningful interactions a *thing* can be involved in, as well as their suitability to interact with other *things* in a smart space.

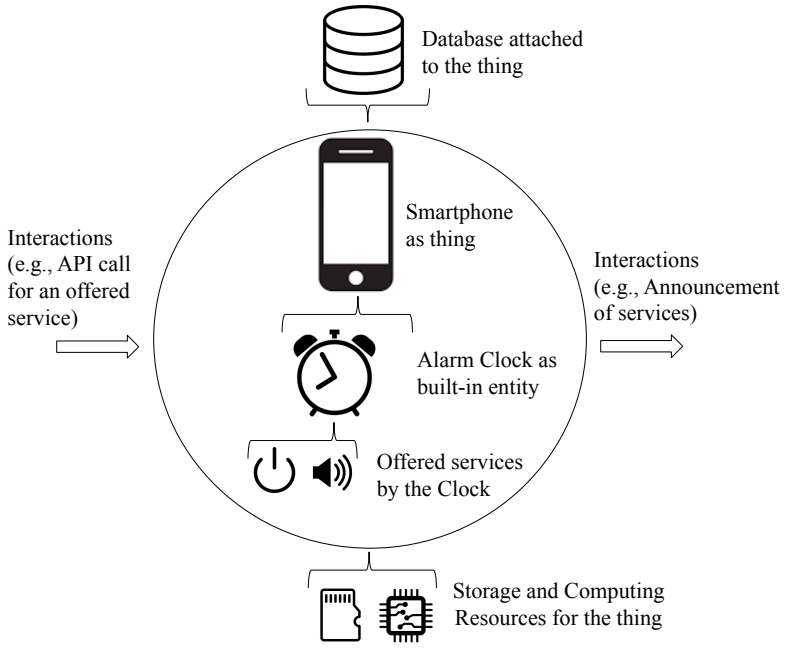

Figure 4-4: The components of an example hybrid thing.

With these differences in mind, the IoT-DDL defines a set of major components that can represent a given *thing*, and uses them to structure the top-level format of the description itself. The components, illustrated in figure 4-3, are:

- *Resources:* the shared components that a *thing* requires to be a part of the IoT, such as the networking hardware, storage device, etc. Each resource can be described by a set of attributes and properties that dictate the operation of essential internal functions.
- *Entities:* individual hardware and software components that represent a core set of functionalities within the *thing* device. An entity defines properties that control how the component operates when used in external interactions.

- *Services:* functions that use the features of their parent entity. A service represents a single point of interaction with the *thing*; it is a well-defined interface (public API) that can be utilized by users or other *things*.
- *Attachments:* a (potentially shared) external resource that can offer additional services to the *thing*. This resource likely exists on the cloud or edge, as it is too heavyweight for a constrained device or requires resources that are not available internally. For example, an attachment may represent a log server, database, or dashboard.

As an example, consider a smartphone acting as a hybrid *thing* in a smart space. The *thing*, illustrated in figure 4-4, contains internal storage and WiFi radio resources, a set of hardware sensor entities (such as the accelerometer), and a software timer entity that can provide a variety of services to the user. The phone also receives occasional operating system updates from the vendor through a cloud attachment. Together, these components make up a complete IoT-DDL description that the *thing* can use to self-identify and better prepare for meaningful interactions.

This description language assists in bridging the gap between the needs and wants of health device vendors, and those of developers and end users. While the vendor has full control over the initial features and behavior of a device, the DDL keeps the *thing* open to changes by other stakeholders as well. A developer can tweak a vendor behavior without needing to completely reconfigure the device; user stakeholders gain more control over their *things* without cutting out the vendor's influence or creating additional work in developing a new device.

The IoT-DDL specification has continued to evolve alongside changes introduced by the Atlas Health IoT Architecture. For example, the architecture uses microservices for *thing* functionality on capable platforms, matched by a more explicit service definition model within the description file. This allows behavior to be specified within the DDL and created or changed at runtime. The specification also introduces new concepts such as *actionable keyword*s (described in section 4.4) and structures for declaring proxy interfaces to external devices, including BLE sensors.

To simplify the creation of these descriptions for vendors, developers, and end users, an HTML builder tool capable of creating, editing, and validating IoT-DDL files is also introduced. These components are discussed in detail in chapter 5.

4.2.2 Thing Tweets

Once a device is positioned to interact effectively with its smart space, it must understand how to communicate with other *things*; the Atlas Health IoT Architecture manages this *inter-thing* information sharing through interactions called *tweets*. Devices can share available information and metadata (collected from the IoT-DDL description) by distributing these small messages to all *things* in the smart space. Each tweet contains an identifier for the originating *thing*, along with a payload such as device information, a service name and API, keywords, etc. Tweets are flexible and adaptable to new health architecture features, such as a smartphone application *thing* searching for services to integrate into its interface (as described in section 4.4). Together, these "broadcast" tweets represent the first stage of a meaningful interaction between *thing* devices.

In addition to the broadcast tweets described above, a device may send a tweet interaction directly to a single *thing.* In this case, the tweet contains an identifier for the target *thing* (likely acquired from a previous metadata tweet) in addition to the payload and source information. Such tweets allow devices to invoke services or respond to requests from other communicating *things*. This tweet type is also used in new architecture features, such as carrying *inter-thing* notifications or formatting proxy interactions. This class of "unicast" tweets represents the second stage of a meaningful interaction—allowing *things* to utilize the information they discover and receive from other *things* in the smart space.

4.3 DIY Health IoT Apps

As discussed in chapter 3, the concept of mobile companion applications that control and enrich their devices is exceptionally important in the personal health IoT domain.

Combined with the software *thing* features introduced in section 4.1.2, mobile apps have the potential to integrate further with traditional health *things* and participate in their high IoTility interactions. However, even amongst software *things*, mobile apps are quite different from normal devices in both form and function. An app's detailed and flexible interface (the smartphone's screen) opens up a variety of interaction possibilities, but utilizing these unique features requires rethinking how a mobile app with *thing* features can interact with other devices in the smart space. This section describes an initial attempt at better integrating mobile apps with democratized *thing* devices.

In present-day health IoT ecosystems, many companion apps are tailored to specific environments and *things* that force the user to stay within the bounds of their silo if the app's full functionalities are to be taken advantage of. Future ecosystems could be much different, where these IoT apps are built by the end user in an easy do-it-yourself (DIY) fashion, tailored to their own smart space and devices. Such capabilities could help limit the "app overload" described in chapter 3 as well; a user with many health devices from different vendors (and therefore many companion apps) could effectively "concatenate" the information they are most interested in, creating a single DIY app containing only this focused information. To achieve this, a variety of questions must first be answered, such as how the app interface can be defined and how the user can design and build a new app directly from their mobile device.

First, consider the following example of a simple DIY health app. A user owns, among others, two health devices: a connected body weight scale and a fitness tracking mobile app acting as a *thing*. The app contains functionality to calculate the user's body mass index (BMI)—under normal circumstances, this requires the user to manually input their current weight (their height is stored within the app's profile) after taking a reading on the connected scale. Both devices sport *thing* capabilities, but there are no pre-existing integrations between the two, limiting their combined use. Instead, imagine the user defines a DIY app relating the scale's API to the BMI functionality of the wellness app. The new app presents a simple interface showing

the calculated BMI after the scale is used, without requiring the user to open the apps for the scale or fitness tracker.

Using the Atlas Thing Architecture, *things* can tweet to share identities, services, and potential relationships [44] that provide hints as to how devices may be related and capable of working together. These relationships empower the user, allowing them to discover new interaction possibilities or create their own while charting out the functionality of a new DIY app. To enable generation of this app without involving the user further, a description of these functions can be sent to an external attachment (a cloud or edge server) capable of converting such information into actual mobile code. The attachment builds an application package, using the APIs of the involved *things* to create a simple interface, before sending the binary back to the user's smartphone; the end result is a native application (versus a "bookmark" or web app link) enabling interactions unique to that user's smart space.

These concepts were utilized in a prototype implementing the DIY health app scenario described above [77]. This prototype partially served as a guide while developing features of the Atlas Health IoT Architecture; it begins to show the potential of more generalized app-*thing* interactions, but also highlights some issues with these simpler interactions. For example, the above works best with sensor *things* and APIs that take no input parameters—complex *thing* services demand complex user interfaces that are harder to generate dynamically. Additionally, the concept depends on the generated app superseding the use of its components. If a user likes or wants other features from the fitness tracker app described above, for example, they cannot easily access these capabilities within the newly generated application. The next section builds on this concept and its shortcomings, introducing more direct *thing*-like behavior and better integrating dynamic opportunities into existing mobile applications.

4.4 Mobile Apps As Things

Imagine a user downloads a mobile app within their smart space full of personal health *things*. For example, consider a COVID-19 tracking and information app that provides advice and contact tracing. Surrounded by a variety of devices with the potential to track symptoms such as body temperature or coughing, the app could offer feedback beyond “do you feel ill?” or “have you been tested?” prompts. The presence of such *things* may drive the app to adjust its interface, creating a new button that requests the devices check for a high temperature or listen for continuous coughing, and provide deeper feedback to the user.

While a vendor may be able to achieve this within their own ecosystem silo, imagine the interaction taking place without programming the specific interaction—the app simply advertises a set of capabilities and interests that allow *thing* devices to create new engagement opportunities. The *thing* devices described above are looking for these capabilities, allowing them to drive a new interaction on the mobile app without the original developer implementing or even considering this specific case of a “COVID check” button. However, this raises a variety of concerns: how should the app present these dynamic opportunities to the user, and how can the developer prepare their app logic and interface for such opportunities?

To act as a thing, a mobile app must react to the capabilities and opportunities of a smart space, and change at the will of its user within the context of its own interface and functionality. This calls back to introspective programs [78], or programs that "self-reference" their own information. However, instead of performing tasks like copying itself or print out its source code, a mobile app must use this information for far more practical purposes, such as adjusting their services, appearance and functionality on the fly to take advantage of and utilize the services of another smart *thing*. Before such modifications can occur, two pieces of information must be known: the target functionality (from the smart thing), and the control to give it (from the mobile app). Once both of these are known by one of the devices, the interaction and app augmentation can occur.

Receiving potential functionalities from the thing through the transmission of its APIs, as was done in a previous *apps-as-things* demo [79], provides an app with all the information it needs to form new meaningful interactions. However, this method also leaves a lot of work for the app developer, in terms of determining how to integrate this new behavior and display it to the user. If the developer does not know exactly which smart things to support, it becomes impossible to determine what context (when and where in the app) an interaction should be made available. In this manner, the app developer may want to leave some "placeholder" space for a new interaction's UI elements; however, anticipating the extent of the requirements (mainly where to place this placeholder and how to link it logically with the created new behavior) would prove difficult. Instead, a UI in this manner would likely find most use in pop-ups (such as a toast notification [80]), or a new interface in its entirety (such as a "Chromeless" in-app web browser view [81]), where the context of the interaction can be entirely contained. This moves the information requirement to the *thing*, which can now manage interactions through its own interface, using the mobile app only as a display. Unfortunately, in these cases, this context is disjoint from the rest of the interface: the UI exists "on top" of the app and cannot easily interact with or extend the app's developer-created elements.

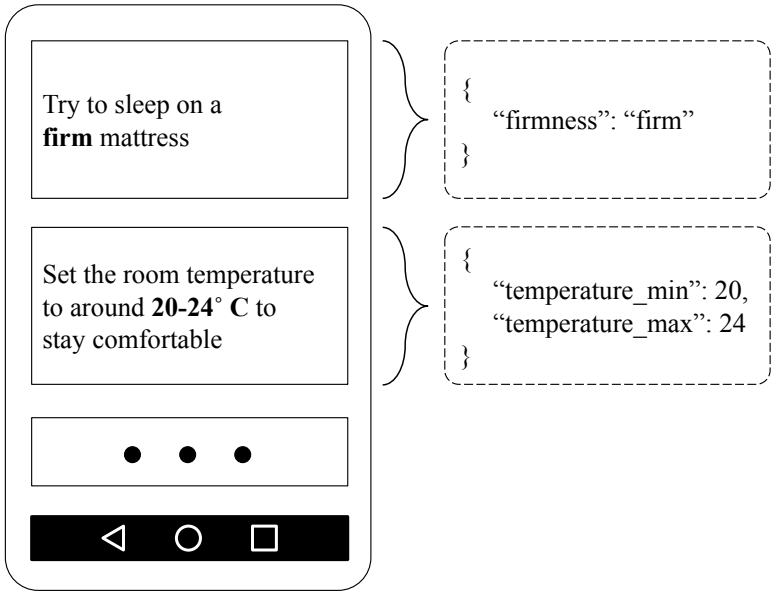

Figure 4-5: Contextual information available within a mobile app.

True integration of a *thing's* functionality into a mobile app, therefore, becomes difficult. While handling "simpler" interactions this way may be possible, such as when a sensor value should be displayed or when a button can trigger a service with no arguments (as in the above example), it becomes less feasible when handling services that require input from the user. Here, both the *thing* and the app lack the full picture of the interaction; the app does not know what information is needed by the *thing* (in terms of an API call), but the *thing* cannot pull arbitrary information from the app. Instead, more complicated interface decisions would have to be provided directly by the *thing* (in the Chromeless web view, for example), asking for information that may already be available within the app. If the app developer only knows about what information can be provided to potential things, and the *thing* only knows the services it can offer, how can interactions between these devices be more closely integrated?

4.4.1 Using Context in Mobile Apps

Consider, for example, a pain management app and its developer. The app presents a variety of tips and instructions for the user to manage different forms of chronic

pain, including a one layout with sleep tips such as “use a firm mattress” and “sleep with a room temperature between 20-24˚ Celsius.” Imagine that, like the concepts above, the app reacts to the user’s smart space, showing new buttons to set the firmness of the user’s smart mattress or schedule a temperature on their thermostat. Rather than explicitly program these interactions, the app developer merely exposed key concepts, such as “firm mattress” or “22˚ C” to the smart space, and the engagement opportunities are initiated by the *things* that could use this data if they existed in the smart space.

In this manner, various “contextual” data from the app can be quite useful in integrating new smart *thing* interactions. Even without knowing what devices to interact with, the developer already knows where interesting data comes from (such as a specific UI element), and, by extension, how such data may be used in a new interaction with an interested *thing*—this relation is shown in figure 4-5. In a situation targeting a specific device, the app developer indirectly identifies such “interaction-capable” contextual information to pass to the known API. For example, if the app developer works for a smart bed manufacturer, they know “firm” is the value to pass to the device. When the target functionality is unknown, the developer can still identify this information, effectively saying, “the user is interested in a firm bed” rather than “the user wants to set their smart bed to firm.” While they cannot give this information directly to a *thing* that will use it, the app developer can still provide this input information if a *thing* can relate its services and complete the other half of the engagement.

From the *thing’s* point of view, the app reveals not only a potential place where it may be used, but also the data needed to use it. Even with the right information, however, the *thing* does not know when to use it. For example, the pain management app should not set the bed’s firmness immediately upon starting; the user must see the suggestion and explicitly ask for the change to be made. The app also needs the context of the interaction, or where in the app’s logical flow the data will enter and be made available. In the case of a mobile app with many views and functionalities, such context is critical in enabling meaningful interactions; at times, a

service or a piece of data may not even be relevant or accessible, depending on the state of the app and the actions of the user. Conveniently, knowing where the data comes from provides the contextual information needed: the source UI element is always tied to a specific point in the app's logical flow. For example, the ability to set the mattress to firm only makes sense when the user is looking at the sleep tips view. Therefore, the final "trigger" for the interaction (a UI element; a button in the example) must be located in the same context as the data that will be provided.

This concept is similar to the more traditional Android URI Intent [82]. Here, the developer specifies a format that arbitrary text can be matched against, creating a "hyperlink" to a different app that can use the matched text to perform a service. For example, Android can detect common date and time formats in plain text, which are automatically underlined to create a link. When the user taps on this link, they are given the option to add the event to their calendar, amongst other operations. Here, the original writer of the text was not concerned with this functionality; they did not write HTML tags to specifically allow the time to be used in the calendar. Like the concepts above, the "consumer" of the interaction (the *thing*, or the Android OS in this example) decided how to use the data.

Unlike a date URI, however, useful app information is unlikely to follow a standard format. Instead, collecting and presenting this per-component information poses a significant challenge to the developers of both *thing* devices and mobile apps. Making such fine-grained interaction a reality will require mobile app developers to truly consider the role of their app in an IoT system, manually identifying pieces of information that could be used, for some purpose, in a smart space. Likewise, *thing* developers must support a wider range of potential interactions, looking for key pieces of data that are compatible with their services, making their *things* more usable and accessible through relevant mobile apps. Through a new programming construct introduced in the next subsection, this thesis aims to limit such apparent complexity by minimizing requirements placed on the mobile app developer—where a mastery of IoT should not be required to create an app with these *thing* capabilities. A solution also should impose only minimal

changes to the app development process; even with IoT knowledge, app developers are less likely to go out of their way in making their apps *things*. Rather, the capabilities introduced should exist as a first-class citizen within the standard development ecosystem.

4.4.2 Actionable Keywords

For a mobile app to better realize its functionality as a *thing*, the roles of a mobile app and normal *thing* device are reversed. Rather than search for potential interactions through service/API information broadcast from *things*, the mobile app advertises its available input data back to the smart space. A *thing* can then match this input against its available services, discovering new potential interactions that can be shared with the mobile app. With knowledge of the input data, the mobile app developer can directly design an appropriate interface (versus leaving space for the Chromeless web view described above) while remaining flexible for various potential interactions. This also enables other *thing* devices to make the decision on actual matches and have more means to interact with the mobile app.

The actionable keyword (AKW) concept and programming construct can be used to represent this input information and facilitate a link between the logic of an engagement opportunity and the mobile user interface. An AKW—as shown in figure 4-6—is a structure present within a mobile app that contains the following: a description of some input data, a set of keywords to represent the purpose and source of that data, and information on the UI elements that data is tied to (the context of the interaction). Portions of this information from the AKWs are then broadcast to the smart space, where *thing* devices compare keywords and data types against their available services. If a match is found, that service's API is sent back to the app to be invoked by the user. Note that a single AKW may represent multiple instances of such a UI context (like the rows of a list view); the actionable keyword represents the type of information available, not a specific piece of data—all instances of that data are valid inputs to a thing service that matches with the AKW.

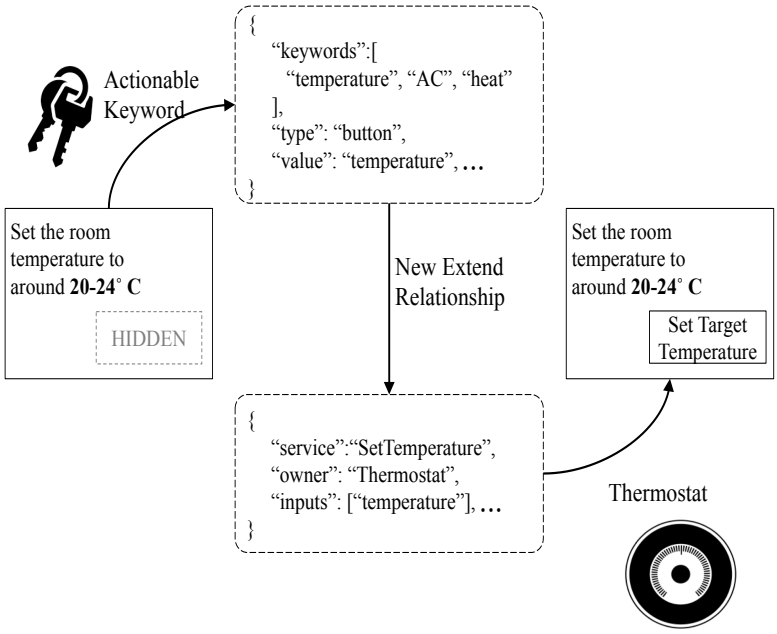

Figure 4-6: The interaction between an app's actionable keyword and a thing service.

The AKW also handles another important aspect discussed in the previous subsection: once a relationship is formed, when should the data be used to invoke the bound *thing's* functionality? To address this, an actionable keyword also represents an input control (also called a widget [83]) within its context that allows the user to trigger the formed relationship. The developer defines a "placeholder" element (a button in the current implementation) that is initially blank and hidden—it is not known what service, if any, will respond to the AKW (if nothing responds, the element will stay hidden indefinitely). Once a relationship is established, the button pops into view with a label or styling specified by the binding *thing*. At this point, interacting with the element (such as tapping the button) invokes the received *thing* service with the data from the relevant UI context as input.

To illustrate this entire process, consider the pain management app described above. The app developer believes smart *things* could help the user better act on the various suggestions within the app, and decides to transmit the ideal temperature for sleeping. They create a new AKW, specifying the temperature as an available integer, and relating it to the keywords "temperature", "air conditioner", and "heater," among others (since the user may have central air, a radiator, or even a

fan, the developer does not know exactly how temperature can be controlled). Within the app's UI, the developer specifies the text view as the context of the potential interaction and positions a placeholder button next to the text itself. All of this information is compiled into the app, and broadcast to the smart space while the app is open. These broadcasts can then be evaluated by other *things* in the smart space, which attempt to match the keywords and data types with the capabilities of their own services.

A thermostat *thing* finds a match and responds to this AKW to create a new relationship with the app, providing information about itself, its "set temperature" service, and a new label for the placeholder button within the app. The app can then concatenate its ability to provide a temperature with the *thing's* service, enabling the placeholder button and changing it label to "Set Target Temperature." When the button is tapped, the service provided by the thermostat is invoked with the temperature specified in the sleep tips. Note that in this scenario, multiple services may vie for an AKW, but only one may attach and create a relationship—this, however, is not permanent; the bound service may be removed, disabled, or swapped out, allowing another to match.

In the given scenario, all information needed by the thermostat service is available from the app immediately, and the user can complete the interaction with the single tap of a button. The interaction involves a *thing* requiring a specific input type, receiving the relevant AKW, and instructing the mobile app to integrate its service. However, this kind of back-and-forth may not always be able to satisfy a thing service's requirements, especially with more complicated services, such as those requiring multi-stage interactions or user confirmation. Handling such interactions is likely to drastically increase the complexity of the solution, therefore, the proposed systems focus only on "simple" interactions utilizing a single button.

4.5 SUMMARY

This chapter described the design of a health IoT architecture influenced by the requirements presented in chapter 3. The Atlas Thing Architecture, focused on personal IoT applications, was used as a base for introducing new, health-specific features and requirement implementations. The Atlas Health IoT Architecture, whose structure was broken down in this chapter, reflects the accumulation of these new features. Most of these features reside in one of two major subsystems: the *Lower Health IoT Layer* and the *Higher Health IoT Layer*, resting between an abstraction layer and a communication interface. The higher health IoT layer provides self-contained implementations derived from the requirements (such as IoTility and Safe Use), and augments interactions that occur in the lower health IoT layer—which focuses on core functionality such as handling API calls received from other *thing*s—as they are sent and received throughout the smart space.

This chapter also introduced some features utilized by the architecture to better support democratization, including "tweet" communications and the IoT Device Description Language (IoT-DDL). Tweets are text-based messages that are used to transmit information about a *thing's* services and capabilities, invoke functionality on a device, or flexibly enable new health interactions. The IoT-DDL is a human-readable configuration schema that allows vendors, developers, and end users to configure and describe the capabilities of their various devices. Like tweets, the IoT-DDL is also extensible and has adapted to new health IoT features and requirements alongside the *thing* architecture.

Lastly, the chapter introduced some IoTility concepts that play a role outside of the core *thing* architecture, defining new interactions with mobile device apps. DIY Health IoT Apps allow end users to combine functionality from independent *thing* devices (utilizing their public APIs) into a simple interface that can be installed as a native app. Mobile Apps As Things (MAAT) evolves this concept further, allowing developers to specify individual "placeholder" UI elements that can be linked to new *thing* device behaviors. These placeholders are created in the same manner as

normal interface elements and associated with *actionable keywords* (AKWs)—containing keywords and potential arguments for an IoT service call—that allow the developer to specify potential behaviors/features without programming for a specific *thing* or class of devices. Instead, the *thing* makes this matching decision, shifting responsibility and simplifying the mobile developer's role in new IoT interactions.

Overall, this chapter outlined the structure of the complete Atlas Health IoT Architecture, especially the components related to Device IoTility and Democratization. This also included considerations for these ecosystem components, such as external services and companion mobile applications. The implementation of the architecture and ecosystem is explored in further detail in chapter 5, and then evaluated through a set of experiments in chapter 6. Some architectural features that received less focus in this chapter, such as those relating to User Identity and Privacy, are also a part of the limitations and future work discussed in chapter 7.

CHAPTER 5: IMPLEMENTATION

This chapter presents the implementation details of the health IoT architecture and supporting components detailed in chapter 4, each aiming to satisfy features of the requirements presented in chapter 3. While the requirements cover a wide area of concerns regarding personal health, this implementation focuses mainly on the issues of democratization and IoTility features (although some parts of the architecture begin to address the other requirements). Section 5.1 implements the Internet of Things Device Description Language, a key element in addressing democratization concerns. Section 5.2 implements the DIY Health IoT Apps concept, making use of some of these democratization capabilities. Section 5.3 implements the various components of Mobile Apps As Things and actionable keywords, a significant enabler of IoTility. Finally, section 5.4 implements a variety of miscellaneous *thing* architecture components that are relevant to the requirements.

5.1 INTERNET OF THINGS DEVICE DESCRIPTION LANGUAGE

The Internet of Things Device Description Language (IoT-DDL), introduced in chapter 4, is an XML-based schema used to describe the capabilities of a *thing's* hardware to the health IoT architecture. This includes metadata and behavioral details that, in addition to internal use, can be broadcast to the smart space to share information and set up new interactions. The IoT-DDL is a core element of the democratization features discussed in chapter 3. Providing a base for many IoTility concepts, it is used to configure the various architectural components detailed in this chapter, such as proxy interfaces, constraints, and MAAT. The following subsections detail the structure of the IoT-DDL, along with a user-facing tool designed to simplify the development of new description files. Additionally, the complete underpinning schema is presented in appendix A.

5.1.1 STRUCTURE

At the top level, the IoT-DDL begins by defining various sets of metadata, as shown in figure 5-1. The first section, *Descriptive Metadata*, describes the *thing's* manufacturer properties, such as make, model, and vendor that can be shared with other devices, along with user-specific details such as location and smart space information. This section also defines the *thing's Atlas Thing ID* (ATID), specific to the device, and *Smart Space ID* (SSID), shared within the user's smart space, to facilitate communication between *things*. The combination of these identifiers provides a unique address to specify the device when sharing information or making service calls within the smart space (as used in the tweet communications described in subsection 5.4.1). On the other hand, the *Administrative Metadata* section exposes information that can be used to initialize the OS components of the *thing* device internally. For example, this includes networking hardware (such as WiFi or Ethernet), access information (for example, SSID and password for WiFi), and protocol information (such as the multicast group for IP-based communication). Note that not all fields of these metadata sections must be defined, and some may not be used depending on the environment of the *thing* device.

The top-level IoT-DDL also defines two lists: one of entities and one of attachments. An attachment, as introduced in chapter 4, defines a service or feature that requires resources and capabilities less likely to be present within a *thing* device. In this thesis, an attachment represents a specific type of resource (such as the app generator described in section 5.2) that can be interacted with through a well-defined API already understood by the *thing* architecture. Within the schema, an attachment is represented by its type and URI endpoint (such as an IP address).

```
<IoT_DDL>
    <Atlas_Thing>
        <Descriptive_Metadata>...<Descriptive_Metadata>
        <Administrative_Metadata>...</Administrative_Metadata>
    </Atlas_Thing>
    <Atlas_Entities>
        <Entity>
            <Descriptive_Metadata>...</Descriptive_Metadata>
            ...
            <Services>
                <Service>...</Service>
                ...
            </Services>
            <Relationships>...</Relationships>
            <Unbounded_Services>...</Unbounded_Services>
        <Entity>
        ...
    </Atlas_Entities>
</IoT_DDL>
```

Figure 4-1: The high-level structure of the IoT-DDL sections.

An entity, the other list element, is a flexible concept describing a single unit of hardware, software, or hybrid functionality that is part of the *thing* device. For example, a hardware entity may represent a single temperature sensor built into the device. Each entity holds a section of *Descriptive Metadata*, similar to that of the top-level Atlas *thing*, containing properties such as name, description, model, and vendor. Additionally, entities may declare a proxy interface (detailed in subsection 5.4.3), such as an external BLE address and its attributes of interest, for interacting with low-power devices and exposing their features to the smart space. Mainly, however, an entity declares the structure of its APIs and the interactions it can take part in, though sets of the following elements:

- *Services:* A service defines a single API endpoint for interacting with its respective entity. This section also defines metadata, including name, description, and keywords, which can be used to filter services when searching (such as the case in MAAT) or with unbounded services (described below). To declare its API, the service also defines its input and output types,

each with a name and optional description. Finally, the service declares a "formula," or short set of instructions to be invoked with the service. The implementation represents this as a C-code excerpt, which is compiled at runtime on a capable *thing* device to formulate its API. This "just-in-time API-ing" (discussed further in subsection 5.4.2) is important to the IoT-DDL's flexibility; rather than the API being part of its compiled firmware, the device uses basic information (input, output, etc.) to create a service definition appropriate for the smart space it exists in. An example is shown in figure 5-2.

- *Relationships:* A relationship represents a logical bond between two services, relationships. While relationships are not acted upon directly by a *thing* device, they act as metadata guiding the creation of new IoT interactions and applications, such as the DIY Health Apps described in section 5.2. A relationship can represent a variety of cooperative or competitive interactions [44], such as *control*, a basic trigger-action linkage, and *drive*, which additionally passes the output of one service as the input of another.
- *Unbounded Services:* With the above elements, a *thing* can only define relationships between services within its own entities—however, most interesting interactions likely exist as relationships with other *thing* devices. To allow an element to reference services that may not exist, an unbounded service can be defined. Such a service represents a "wildcard" that can be matched to a concrete service at runtime based on its metadata. These match requirements can be unique, such as a service from a specific vendor or one with a specific identity, or generic, using keywords along with input and output parameters (similar to MAAT's method in section 5.3). In this case, the required similarity is defined as a "match value" threshold.

```
<Service>
    <Name>GetTemperature</Name>
    <Keywords>temperature, body, thermometer>
    ...
    <Inputs></Inputs>
    <Output>
        <Name>Temperature</Name>
        <Type>float</Type>
        ...
    </Output>
    <Formula><![CDATA[
        unsigned char tl = /* read sensor low bit */;
        unsigned char th = /* read sensor high bit */;
        return (float)((th << 8) | tl) / 100.f;
    ]]></Formula>
</Service>
```

Figure 5-2: The basic structure of an IoT-DDL service definition.

The IoT-DDL also defines various *constraint* fields to enable safe and proper use within the *thing* device. At the entity level, these specify *temporal constraints,* or the time periods when the device can be used (for example, a sleep sensor should only operate in evening hours). Services offer similar constraints, defining the minimum period of time between consecutive API calls and the ability of the API to accept simultaneous invocations, in addition to per-input minimums and maximums to validate passed API parameters. Together, these checks must execute successfully before a service's actual behavior can be run. This can protect against the various scenarios described in chapter 3, such as unexpected behavior or damage to the device due to user error, programming bugs, or malfunctions.

5.1.2 IoT-DDL Builder Web Tool

In practice, an IoT-DDL description is most likely to be first created and shipped by the *thing's* creator; that is, the original equipment manufacturer (OEM) or vendor. By the time the device is in the hands of the user (or developer), its IoT-DDL contains a default set of metadata, services, and constraints that it can use to interact safely and effectively in a smart space. This is not necessarily the end of the IoT-DDL lifecycle, though: as the format is open and human-readable, it can be expanded and

modified by third-party developers or end users to meet more specialized needs. However, using the schema directly may be a daunting task for the average end user without documentation or external help. Instead, the IoT-DDL Builder tool aims to simplify the creation and modification of these IoT-DDL descriptions by organizing the schema's fields and options into a user-friendly web form GUI.

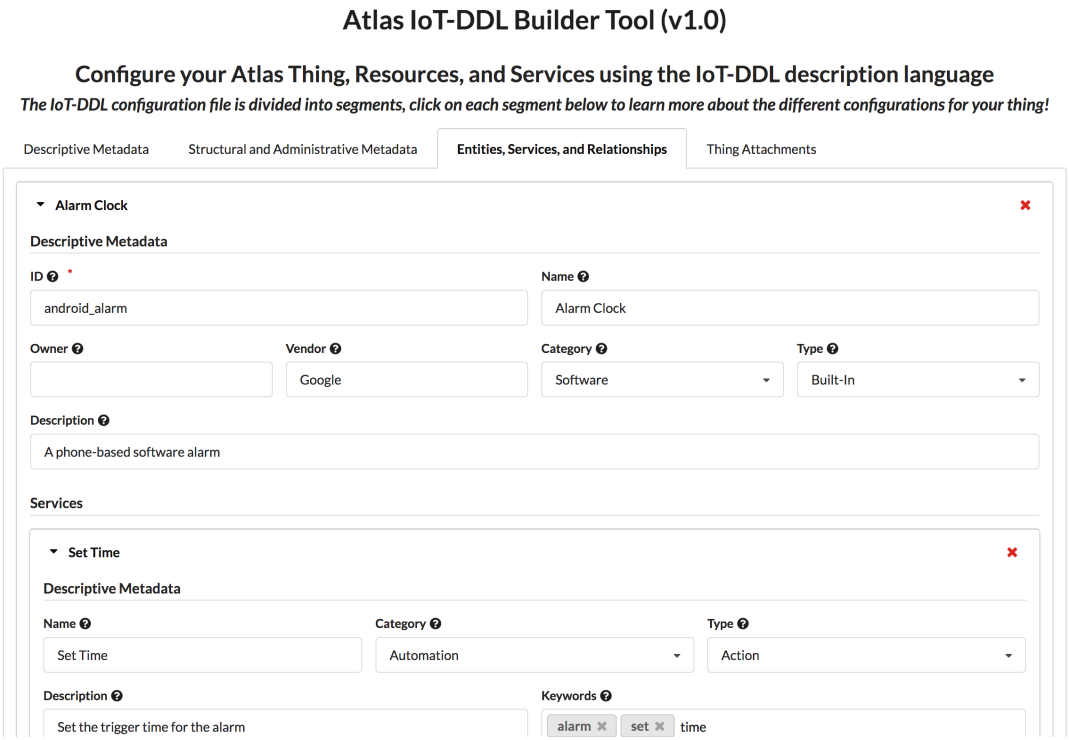

Figure 5-3: The IoT-DDL Builder tool interface.

The builder tool is a single-page HTML and JavaScript web app developed using Semantic UI [84] that can use its form data to create, validate, and export IoT-DDL files. The tool, shown in figure 5-3, consists of four main "tabs" representing each top-level element (Descriptive Metadata, Administrative Metadata, Entities, and Attachments). Each tab organizes its fields to emulate the structure of the schema, providing a mostly one-to-one mapping. The fields reflect their expected data types (for example, token fields for keywords, pickers for dates, and dropdowns for enumerations), and set up hover tooltips next to each label to provide additional help, descriptions, or examples. The Entities tab is the most complex, using an accordion-style nested list to allow users to create new instances of entity, service,

and relationship fields as needed (depicted in figure 5-4). Device APIs can also be written directly in the builder tool: in addition to metadata, a service section allows the user to define inputs, outputs, and functional behavior using an integrated Ace [85] code editor.

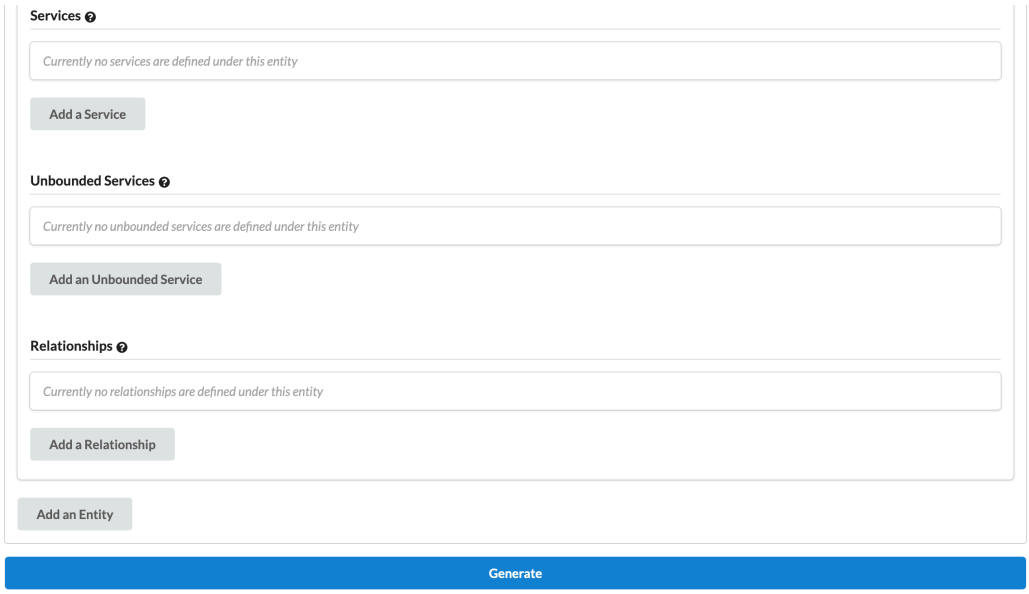

Figure 5-4: A new entity section in the builder, with service and relationship sections.

Once the user fills out the form, the tool can validate elements of the description before generating the final file. Required fields are marked as such, and will raise an error if left empty by the user—some of these checks may also be conditional, such as an SSID and password being needed only when "WiFi" is selected as the networking module. Similarly, the builder can ensure the formatting of special fields is correct, including numbers and IP addresses. After validation, the IoT-DDL is generated by collecting the form fields into a JSON object and converting it to the proper XML format, presenting the completed downloadable file to the user.

5.2 DIY Health IoT Apps

The DIY Health IoT app development concept, introduced in chapter 4, utilizes the democratization features of the health architecture along with the IoT-DDL (detailed in the previous section) to create new Android apps utilizing existing *thing* services

on the fly. These apps offer a simple interface allowing the user to “concatenate” functionalities from different *thing* devices, specially tailored to their smart space and needs. To maximize utility and ease of use, the user should be able to define this functionality and generate the app directly from their mobile device. This section describes an implementation that meets this goal, focusing on a cloud-based service that can compile new IoT apps from a simple description provided by the user.

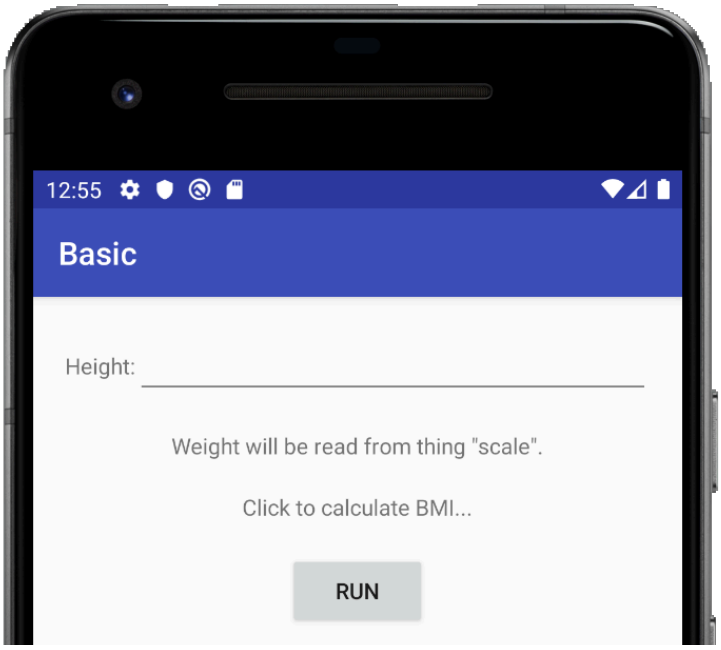

Figure 5-5: The interface of a simple generated app.

Before choosing functionalities for a new app, the user must first be able to identify interaction possibilities in their smart space. Utilizing the service and relationship metadata tweets described in subsection 5.4.1, the user can visualize the capabilities of their smart space (through a special app implemented alongside the demo [77]) and select a subset to be made accessible and usable in their new app. These selected services will be executed in parallel, returning all results to the user (for example, the measurements from a pulse oximeter and temperature sensor will be concatenated into a single result, but the two devices do not interact with each other). More advanced functionalities are represented by relationships, which chain services together by executing them in series and passing their results along.

Consider the example from chapter 4, where the results of the body weight scale's service are passed as one of the inputs to the fitness app's BMI service. These combinations of services and relationships allow users to represent a wide range of potential IoT apps and interactions.

```
<app name="Calculate BMI">
    <relationship name="MyBMI" type="drive">
        <service name="GetWeight" thingid="scale@mysmartspace">
            <output name="Weight" type="float" />
        </service>
        <service name="GetBMI" thingid="wellnessapp@mysmartspace">
            <inputs>...<inputs>
            <output name="BMI" type="float" />
        </service>
    </relationship>
    ...
</app>
```

Figure 5-6: The XML manifest of the app shown in figure 5-5.

Once the user selects the interactions to be included in their app, this information is transferred to a cloud-based service where an Android application binary *(.apk)* is compiled. The generator service accepts an XML-based manifest (an example is shown in figure 5-6) describing the desired services and relationships, and uses it to modify an IoT app skeleton (a basic Android application project). The implementation achieves this through a Python-based templating engine that fills in the needed parts of the app, including the Android manifest (the application name, etc.), the user interface layout, and the main activity logic that invokes the services in the proper order. Note that, as relationships are handled as metadata on *thing* devices, the app itself handles calling the inner services based on the rules of that relationship type [44]. The interface is represented within an Android *LinearLayout* [86] (for a single column form-style input) and uses an appropriate widget type (text field, slider, etc. based on the input type) with a label specified by the IoT-DDL input descriptions for each input. The app functionality is initiated with a button placed below the fields,

while results are displayed in a popup. A generated app, such as that shown in figure 5-5, can then be built, downloaded, and installed directly to the user's mobile device.

5.3 Mobile Apps As Things

Treating mobile apps like traditional *thing* devices is another important concept that can facilitate high IoTility interactions within a smart space. This section describes the implementation of the actionable keywords (AKW) programming enabler introduced in chapter 4, which directly works towards this goal. In addition to new inter-*thing* interactions, fully supporting this concept requires careful consideration of how a mobile app's interface might expose such new features, as well as how a mobile developer can empower their non-IoT apps with minimal difficulty or confusion. The following subsections discuss the AKW framework from each of these perspectives, detailing the *thing* runtime, mobile application framework, and a utility that assists ordinary developers in adding these *thing* capabilities to existing app development projects. Amongst the wide variety of mobile frameworks and development environments available today, the implementation below focuses specifically on the Android platform and the Android Studio IDE [87].

5.3.1 Thing Architectural Components

Within the Atlas Architecture health IoT layers, supporting *thing*-like mobile app interactions requires a *thing* device to listen for AKWs from the smart space and compare them against the services it offers. If a service matches the purpose and API parameters specified in the actionable keyword, the *thing* device can request that the mobile app "bind" to its service; this results in the appearance of new user interface elements within the app and allows the user to invoke the bound service directly. The mobile app is responsible for collecting the required input parameters of the service, which is called through a normal service invocation tweet (detailed in subsection 5.4.1). By utilizing the same method as a normal service interaction, any additional complexity on the *thing* side is minimized, and the AKW lifecycle is

contained within the mobile app logic (for example, if an actionable keyword is "un-bound," no additional action needs to be taken from the perspective of the *thing* device). These steps of advertising, binding, and invoking are illustrated in figure 5-7.

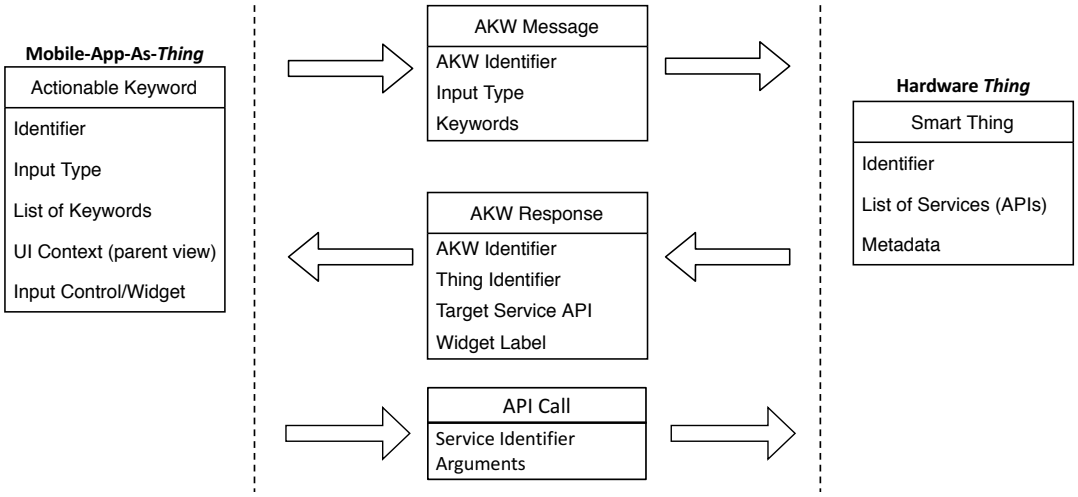

Figure 5-7: The interactions between a traditional thing and an app-as-thing.

In this interaction, the main responsibility of the *thing* device involves comparing keywords and input parameters with its available services. A received AKW, whose simplified JSON structure is shown in figure 5-8, contains a set of input types and names along with a set of keywords that relate to the desired interaction; these correspond to the service keywords and input descriptions defined in the IoT-DDL of traditional *thing* devices (specified in section 5.1). On initial receipt, the architecture uses the input types to filter out incompatible service APIs, and then compares the remaining services' metadata with the AKW's descriptive keywords. On sufficiently powerful devices, the two sets of keywords are compared with WordNet [88, 89], a lexical database that groups words into sets of synonyms and can provide a semantic similarity measurement between a pair of keywords. The required level of similarity can be controlled through a "match threshold" value specified in the service's IoT-DDL information: the architecture uses the Inverse Document Frequency (IDF) [90] measure to quantify the uniqueness or importance of individual keywords. Other more constrained *thing* devices can still participate in AKW interactions by directly

comparing keywords for equality. With the assistance of the developer plugin and keyword repository detailed in subsections 5.3.3 and 5.3.4, the chance for exact matches is greater: the developer can select keywords that are already present in the IoT-DDLs of existing *thing* devices.

```
{
    "type": "object",
    "properties": {
        "Tweet Type":  { "const": "AKW"    },
        "AKW ID":      { "type" : "string" },
        "Input Types": { "type" : "array", "items": { "type": "string" } },
        "Input Names": { "type" : "array", "items": { "type": "string" } },
        "Keywords":    { "type" : "array", "items": { "type": "string" } }
    },
    "required": ["Tweet Type", "AKW ID", "Input Types", "Keywords"]
}
```

Figure 5-8: The abbreviated JSON Schema definition of an actionable keyword.

Once a match is successful, the *thing* device prepares an AKW response for the app. The response contains the information needed to call the selected service—namely, the *thing* identifier and the service name. Additionally, the response specifies a descriptive label that can be displayed within the app interface (in the current implementation, this becomes the label of the placeholder button described in the next subsection). This label can be specified with an additional XML tag in the relevant IoT-DDL service block. Once the response is sent back to the source of the AKW, the MAAT-specific behavior of the *thing* device is complete; the *thing* is already prepared to respond to service invocations from the mobile app (or other devices).

5.3.2 Mobile Application Library

To behave like a *thing* device and support the actionable keywords concept, a mobile app must be able to communicate with the smart space, interact with other devices, and bridge these elements with its non-*thing* components such as the user interface. Additionally, with the possibility of multiple apps acting as *things,* the mobile device itself becomes a "*thing-of-things,*" and must manage distributing interactions to its

child *things* (apps) as appropriate. To meet these needs, this implementation introduces an Android background service [91] to run the *thing* architecture components (to broadcast and receive AKW information), and a supporting framework allowing developers to expose these new capabilities in their apps (to augment the user interface with available *thing* services).

On the mobile device, the background service assumes the role of the Communication Engine in the traditional architecture. With the possibility of multiple apps-as-*things* on a device, a single application should not monopolize interaction with the smart space by networking on its own. Instead, one instance of the background service is shared between all apps, listening for incoming tweets and routing interactions to their appropriate destinations as Android IPC calls. In the context of actionable keywords, this service maintains a list of an app's active AKWs, listens for binding responses, and sends API invocations when requested. However, the service is also capable of collecting metadata from the smart space or forwarding incoming service calls in more generalized app-as-*thing* interactions. The lifecycle of the service process is tied to the longest-living MAAT app; that is, it does not need to provide *thing* functionality once no *thing*-like apps are running.

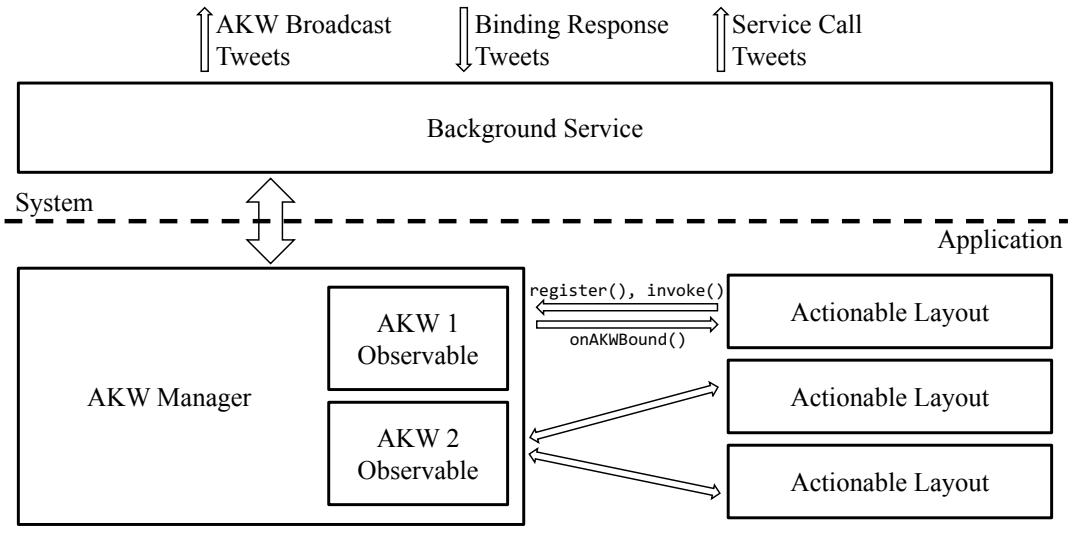

Figure 5-9: The components of the Android actionable keyword library.

Within the apps themselves, the actionable keyword library provides a custom "Actionable Layout" UI component that controls the appearance of the AKW input control, as well as an "AKW Manager" class that handles connecting to the background service and tracking the state of the layouts. These roles are illustrated in figure 5-9; Actionable Layouts register themselves with the manager, which uses their information to create AKWs and set up tweets with the background service. Note that the relationship between AKWs and Actionable Layouts may not be one-to-one in some scenarios (such as the list view described in chapter 4). To handle this, the AKW Manager maintains an Android Observable [92] per AKW that is notified when a binding response is received from the background service. New layouts are automatically subscribed to the appropriate Observable as they are registered with the manager.

In the app's interface, the actual context of an AKW interaction is represented by an Actionable Layout object. This custom view component extends the Android *ViewGroup* class [93] (similar to the standard Android classes *FrameLayout* and *LinearLayout*) and is intended to wrap around an existing view or widget that can expose *thing*-like functionality. Each layout requires the following information from the developer: 1) a list of keywords, 2) the descriptive names of the provided data, 3) the AKW input control (a button component), and 4) a data formatting callback (which converts layout info into JSON arguments), all of which can be specified either programmatically or through an Android XML layout (such as in figure 5-10). The layout places no restrictions or requirements on its child elements beyond the button component specified in the attributes.

```
<ActionableLayout
    android:id="@+id/myAKW"
    app:keywords="sleep,bed,mattress"
    app:data_format="[Firmness]"
    app:button="@id/akwControl"
    app:onFormat="getFirmness">
    ...
    <Button android:id="@+id/akwControl" />
    ...
</ActionableLayout>
```

```
public JSONObject getFirmness(View layout) {
    String firmness = (String)layout.getTag();

    JSONObject jo = new JSONObject();
    jo.put("Firmness", firmness);

    return jo;
}
```

Figure 5-10: An ActionableLayout XML definition with its data formatting callback.

When initialized, the Actionable Layout passes these keywords and input descriptions (whose types can be deduced from the formatting function) to the AKW Manager singleton and sets the child button's visibility to *false*. The invisible button still takes up space in the layout, meaning the rest of the interface remains unchanged. The layout also sets the button's *onClick* callback to a stub function that invokes the specified formatter (shown in figure 5-10) and uses its JSON result to initiate a service call through the AKW Manager. This behavior is important, as the contextual data of an AKW is not necessarily constant, especially when multiple layouts represent the same keyword. The formatter function allows the developer to pull the data from their application logic without needing to interact with any internal AKW states or behaviors. Once initialized, the layout waits to receive a binding notification from the AKW Manager, before making the button visible and setting its label to the string specified in the request.

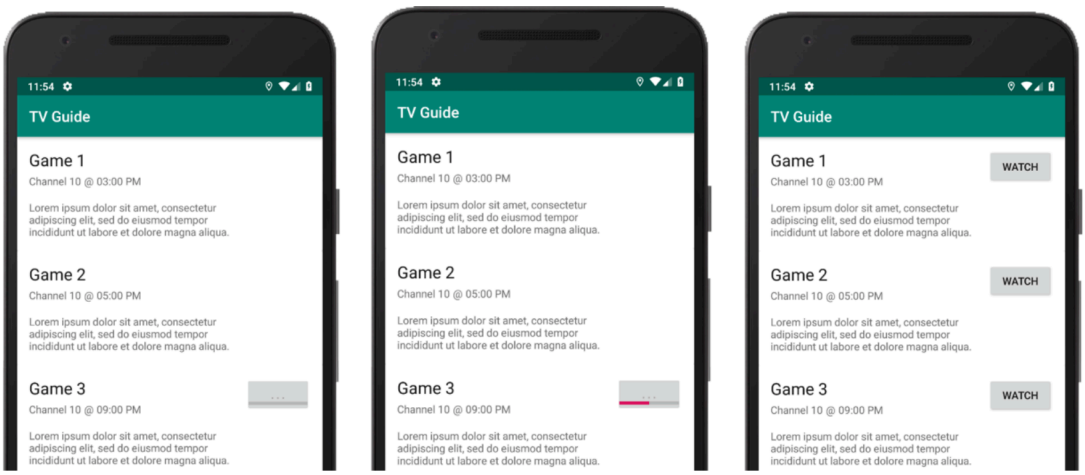

Figure 5-11: The states over time (left to right) of a successful AKW search and match.

The manner in which this button appears is also an important consideration. Given the relatively unusual form of interaction, user understanding is another important part of the actionable keyword concept—especially since the signs of an AKW are mostly invisible before a match is made. Depending on when the match is made, the user may miss interaction opportunities by navigating throughout the app too

quickly, or become confused when a UI element has changed since the last time it was viewed. Even when the user notices the moment of "pop-in," such an event could easily lead to a feeling of "where did this button come from, and why?"

To combat this, the implementation offers an alternative button behavior that overlays a progress bar-style animation on top of the button. In this case, the button is not hidden at the start (but still has no label) and can raise awareness to the ongoing search for compatible interactions. If enough time passes without a match and the progress bar fills completely, the button can then fade away to convey that no AKWs were found. While this method hints to the user that there is additional potential in an area of the UI, it comes with its own set of concerns. For example, a quick match may cause the progress bar to appear only briefly, confusing the user or making them feel like something was missed. Similarly, a completed progress bar (when no match is found) leading to the disappearance of an element may also be unexpected or confusing. While the progress bar has some advantages, this implementation focuses mainly on the simple "pop-in" behavior. Both methods are illustrated in figure 5-11: the first two list elements represent the "pop-in" behavior, and the third shows the stages of the progress bar.

5.3.3 Developer Plugin

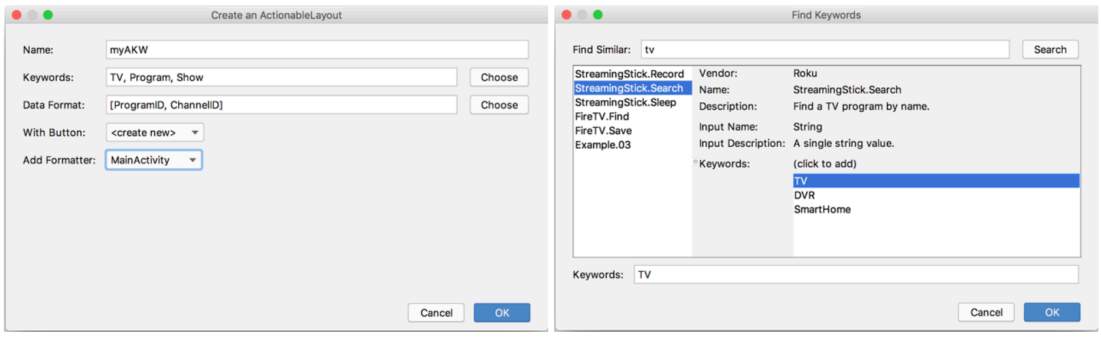

Figure 5-12: The Android Studio plugin interface and keyword search view.

Compared to the first apps-as-*things* demo [79], which used an app-wide set of keywords and capabilities along with interface elements provided by the *thing*, the actionable keywords concept places greater importance on the role of the app

developer. The app initiates these interactions (rather than the *thing* device as in the demo), meaning the mobile developer must consider in detail which elements can be used by the smart space while also describing the keywords and data they offer. Similarly, while the purpose of an app overall may not change over time, individual AKW requirements may need to be added or removed as features change or the UI is altered. These aspects increase the burden on the developer, who may not be very familiar with adding IoT functionality—a serious concern when one of MAAT's goals focuses on simplifying these interactions.

To ease the developer's role and facilitate the creation of actionable keywords, the MAAT framework introduces an Android Studio IDE plugin. This plugin provides support in: 1) creating an Actionable Layout and specifying its fields, 2) choosing appropriate keywords and data formats, and 3) implementing the data formatting callback. To open the dialog, the developer simply selects an existing component with the layout XML that will be converted into an AKW. The plugin registers an *Intention action* [94] (the "lightbulb" pop-up menu shown in figure 5-13) that provides the option to convert the selected layout to an Actionable Layout. In addition to accessing the dialog, the Intention also provides context (a position in the file), allowing the plugin to retrieve buttons within the layout (for the *"With Button:"* field in figure 5-12) and insert new XML tags in the correct location automatically.

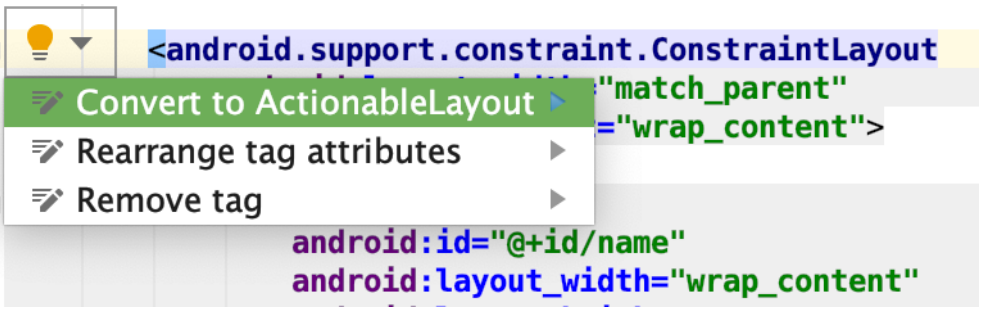

Figure 5-13: The ActionableLayout intention action provided by the plugin.

Within the plugin dialog, the developer can fill out the needed AKW information using text fields or dropdowns as appropriate. The fields for keywords and data types

may also create an additional dialog (using the *"Choose"* button in figure 5-12), allowing the developer to search a repository of existing keywords (described fully in the next subsection) within the plugin and select directly from this information. Accepting the dialog wraps the selected layout element in an *ActionableLayout* tag with appropriate attributes, and creates a skeleton function for the data formatting callback within the specified *Activity/Fragment* class. This skeleton function initializes a JSON object with key names from the input data format, leaving the values to be filled in by the developer.

5.3.4 Keyword Repository

As mentioned above, the developer's choice of keywords and data descriptions is critical to the successful functioning of the actionable keywords concept. For example, consider an extension of the scenario from chapter 4—the developer of the wellness app creates an actionable keyword with a "firmness" parameter, but a certain smart bed device is looking for a "softness" value instead. As long as the app developer is unaware of this specific smart bed and its parameters, a match cannot be made, even though both sides represent an interaction that is effectively equivalent. The keyword search of the AKW plugin aims to resolve these scenarios: imagine the developer searches for bed-related *thing* interactions, sees smart beds using the "softness" value, and adjusts their AKW definition to accommodate and enable the interaction. The keyword search can increase this probability of apps finding successful matches (compared to the developer making up keywords or choosing randomly).

The plugin's search is backed by a cloud-based "keyword repository:" a simple database and web service that indexes *thing* services, keywords, and API parameter information. To populate this database, the repository is capable of scraping fields from existing IoT-DDL files (as described in section 5.1) that are uploaded to it, each containing fields that correspond directly to the information needed by an AKW, as well as human-readable metadata and long-form descriptions for the device and its services. Combined, this information allows developers to

search through existing *thing* information with their own terms and ideas (as shown in figure 5-12), while choosing keywords and data formats that are guaranteed to work with some variety of devices. While the repository does not inform the developer exactly which *things* should be supported by their app, it offers insight into potential *thing* interactions that may influence their keywords, the structure of their app, or the data they provide to better support a wide range of IoT devices.

5.4 THING ARCHITECTURE COMPONENTS

The Atlas Health platform integrates the concepts and features described above into the *thing* architecture and runtime introduced in chapter 4. The full feature set of the architecture, written mainly in C++, targets higher-capability embedded devices and single board computers (such as Raspberry Pi [74] or Intel Edison [95]), but also includes provisions for mid-power devices (such as ARM Mbed [26]). Various compilation flags can disable features such as IoT-DDL file loading (for devices without persistent storage) or microservices (which normally requires dynamic loading), instead using alternatives more appropriate for these constrained devices. This allows the architecture to support a wider range of devices without compromising on its more powerful features.

When an Atlas Health *thing* is initialized, the runtime begins by loading and parsing its IoT-DDL. This description, usually loaded from a file on the device, defines keywords and attachments, sets up API services, describes the device identity, and configures the various higher layer architecture components. A complete IoT-DDL is capable of fully initializing the *thing* device without needing additional information or intervention from the user. For devices without persistent storage (as mentioned above), the IoT-DDL may be embedded directly into the architecture during compilation; in this case, the build script selects a subset of information from a given IoT-DDL file and uses it to replace the parsing procedures with statically generated initialization code. This allows these devices to use an IoT-DDL in the same manner as more powerful *things* (at the expense of recompilation after any changes).

While a *thing* device is running, many of its smaller architectural components also work towards enabling the requirements described in chapter 3. For example, the *Interface and Communication Engine* layer can further facilitate democratization, allowing services to be advertised and invoked through REST and MQTT protocols in addition to the native methods provided by the tweeting system. The following subsections expand on some of these smaller components and their roles in the requirements, focusing on tweets, microservices, and proxy interfaces.

5.4.1 Tweets

Introduced in chapter 4, tweets are the main method of communication between Atlas Health *things*. An individual tweet is represented as a JSON object that can be sent within a UDP packet over the network. The different tweet types extend from the basic format shown in figure 5-14; this includes a tweet type, the sending *thing*, and a timestamp. The sender information consists of the *thing's* ATID and SSID, as well as an optional *thing* name and smart space name for easy identification. The remaining tweet fields are dependent on the specific tweet type, which falls into one of two general categories: "broadcast" information-based tweets, and "unicast" request- or result-based tweets.

```
{
    "type": "object",
    "properties": {
        "Sender": {
            "type": "object",
            "properties": {
                "Thing ID":   { "type": "string" },
                "Space ID":   { "type": "string" },
                "Thing Name": { "type": "string" },
                "Space Name": { "tpye": "string" }
            },
            "required": ["Thing ID", "Space ID"]
        },
        "Tweet Type": { "type": "string"  },
        "Timestamp":  { "type": "integer" }
    }
}
```

Figure 5-14: The abbreviated JSON Schema definition of a generic tweet.

An information-based tweet sends *thing* metadata to the smart space over a multicast IP group. This includes vendor information, descriptions, and keywords parsed from an IoT-DDL, and can refer to the *thing* itself, its services and relationships, or an actionable keyword on a mobile device. On the other hand, a request-based tweet is sent directly to an individual *thing* over UDP to initiate or complete an interaction, including invoking a service, receiving the results of an API call, or binding to an actionable keyword. This category of tweet also includes a *Receiver* object for verification, which follows the same format as the *Sender* field.

5.4.2 MICROSERVICES

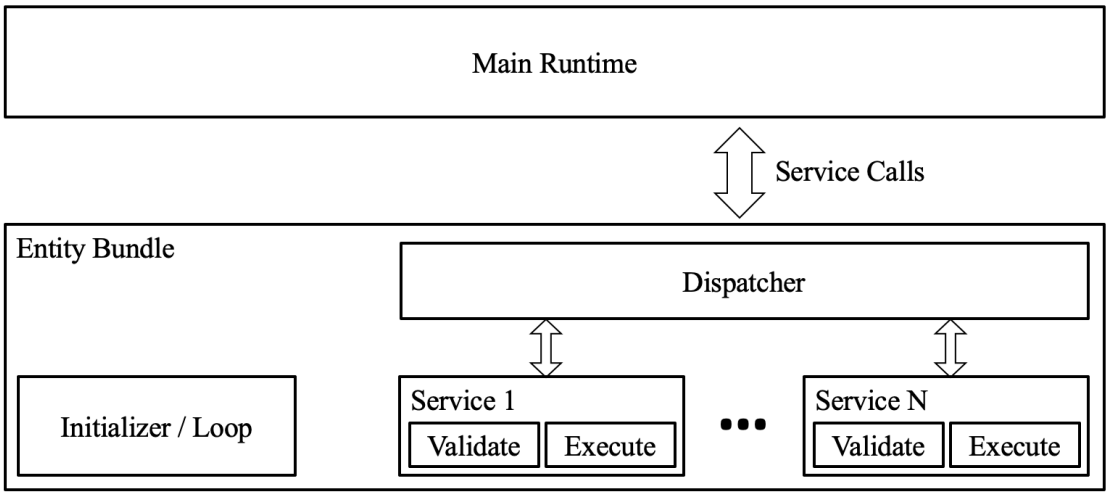

Figure 5-15: The structure of an entity bundle and its interactions with the runtime.

The concept of "just-in-time API-ing"—enabling a service to be dynamically created based on its specification—is an important part of the IoT-DDL design. This decouples the *thing's* behavior from its low-level implementation, allowing service logic to be defined and changed in the description without recompiling the architecture. To support such a feature, the architecture adopts a microservices pattern using the CppMicroServices [96] library. With this pattern, the services listed within an entity are compiled at runtime (directly on the *thing* device, assuming an appropriate compiler is available) into a *bundle*: a dynamically linked binary that exists

independently from the main *thing* architecture executable. This bundle can be loaded, unloaded, and reset on demand, allowing new behaviors to be created on the fly. Each entity bundle, shown in figure 5-15, follows a simple template that ensures the *thing* architecture is able to locate and execute its offered services.

During generation of a given entity, each service defined in the IoT-DDL is translated to a complete C++ function (using its name, parameter info, and behavior), along with a separate "initializer" function defined in the entity's top-level description; when a bundle is started, this initializer can perform one-time setup or configure an update loop as necessary. The generator also defines templated logic to register each service with the CppMicroSerivces context, such that incoming services calls can be dispatched to the appropriate function. Once this code is generated, the architecture invokes a build script that compiles the bundle before loading it into the runtime. After starting the new bundle and creating a thread to run its initializer function, the contained services are ready for interaction; service call tweets are forwarded to an entity bundle based on the received API name, where they can be validated (through another generated function created using the specified parameter ranges in the IoT-DDL) or executed as needed. Note that in scenarios where microservices are not supported, similar behavior can still be achieved by specifying an IoT-DDL at compilation time (as described in the beginning of this section). In this case, code is generated on the build machine in the same manner—allowing it to be statically linked into the architecture binary before loading onto the *thing*.

5.4.3 Proxy Interfaces

Proxy interfaces, introduced in chapter 3, allow a *thing* device to act as an additional host for another device's behavior, exposing and augmenting their APIs or interface elements in a way that could not be achieved alone. This concept is especially important in the context of digital health, where many constrained devices use Bluetooth Low Energy (BLE) and thus only interact with a paired device (such as a smart phone). In the Atlas architecture implementation, a proxy interface

component allows BLE devices to advertise their services to an IP-based smart space through a more powerful *thing* with multiple radios or network interfaces.

```
<Proxy_Interface>
    <Protocol>BLE</Protocol>
    <Address>A8:1B:6A:A8:EC:26</Address>
    <Attribute>
        <Name>TemperatureRead</Name>
        <Handle>0x002A</Handle>
        <Access>Async</Access>
    </Attribute>
</Proxy_Interface>
```

Figure 5-16: A proxy interface with one endpoint specified in an IoT-DDL.

Within the IoT-DDL (detailed in section 5.1), an entity may represent this kind of external device. In this case, a proxy interface can be defined as part of its description, shown in figure 5-16. Within the interface, the device's MAC address is specified along with a list of GATT attributes that should be made accessible to the entity. Each attribute defines a service-like name and the access mode (read, write, etc.), and can be specified by their handle or UUID. During service generation (described in the previous section), the proxy attributes are made into stub functions that interact with the BLE methods of the architecture's interface layer. Integrated this way, normal services inside the entity can then call these methods like normal functions, re-exposing them to the smart space or integrating their functionality.

5.5 Summary

This chapter detailed the implementation of the architectural components introduced in chapter 4, as well as the tools created to support them. First, the structure of the IoT-DDL XML description format was discussed, along with the functionality of the IoT-DDL Builder web application. Together, these components allow vendors, developers, and end users to describe their devices and define new services and behaviors in a standardized format that is flexible and open for later modification. A similar format is utilized in the implementation of DIY Health IoT

Apps, which allows an XML manifest created by the user to be converted into a native Android app through an external web service.

The chapter also described the implementation of MAAT and the actionable keywords programming enabler, both on the device side and the mobile application side. The app developer begins by creating a special layout element around the placeholder elements and context, using the MAAT IDE plugin. This defines the potential input arguments, the descriptive keywords (which can be selected from an online repository built from existing IoT-DDL information), and the interface element that will appear once a *thing* matches and makes a connection. A runtime library and service then handle the rest of these interactions and logic while the app is in use. On the device side, potential AKW interactions are transmitted as tweet messages, and established interactions are handled in the same manner as normal API calls.

Finally, the chapter elaborated on some individual features within the *thing* architecture itself. This includes the JSON format of the tweet messages and the IoT-DDL fields involved in specifying a proxy interface, as well as the details of the API and service subsystem. On powerful platforms, services are represented as microservices, generated from the IoT-DDL and bundled with their parent entity (the hardware or software element that provides their functionality). This allows services to be both specified dynamically and managed independently. On more constrained platforms, the architecture also supports the static compilation of these service behaviors without additional effort from the developer. These implementations, along with those of the IoT-DDL and mobile app features, are used as the basis for the experiments described in chapter 6.

CHAPTER 6: EVALUATION

This chapter evaluates of a selection of the feature implementations presented in chapter 5. Each experiment described below considers an implementation detail of the architecture from a different point of view, from performance evaluation to user acceptability. Overall, the experiments aim to use these architectural components in analyzing the relevance and feasibility of the requirements presented in chapter 3. Section 6.1 evaluates a selection of performance metrics relevant to the actionable keyword feature. Section 6.2 presents a developer case study evaluating the tooling developed for the Mobile Apps As Things concept. Section 6.3 investigates a variety of commercial health devices in order to develop a representative scenario evaluating democratization concepts. Finally, section 6.4 focuses on the performance feasibility of the remaining safe use, IoTility, and identity requirements through a set of representative behaviors running on a prototypical *thing* device.

6.1 EXPERIMENT I: AKW LATENCY AND RESPONSIVENESS

The actionable keyword programming enabler, detailed in chapter 5, introduces a new way to integrate arbitrary *thing* capabilities into existing mobile app interfaces to realize the concept of Mobile Apps As Things (MAAT). When an actionable keyword is bound, the app's interface adjusts dynamically to expose a new *thing* interaction opportunity to the user. As this occurs simultaneously with normal use of the app, metrics such as the processing time of the adjustment become significant: can these interface changes (which may not be anticipated) resolve quickly enough to be effective as the user navigates throughout the app? This section presents a set of measurements to quantify this concept of responsiveness, along with experiments to evaluate an app using AKWs in a variety of representative scenarios. Finally, these results are analyzed to determine the feasibility (in terms of runtime performance) of these dynamic interface changes and the actionable keyword concept as a whole.

6.1.1 GOALS

As mentioned above, the elapsed time between broadcasting an actionable keyword and the appearance of its corresponding interface element is critical. While metrics such as battery usage and processing power are important in all mobile applications, this latency measurement has the greatest effect on the immediate user experience: as each AKW is tied to a specific piece of the app UI (local to a single "view"—an *Activity* or *Fragment* [97] in Android), the overall visibility of a MAAT interaction is temporary (and possibly brief) depending on how the user navigates throughout the app. In a given scenario, an interface change with high latency may be at best confusing or annoying, and at worst missed entirely. Therefore, the time needed to broadcast and process an AKW should be minimized for an optimal interaction.

However, defining a maximum acceptable latency is not straightforward; the exact duration that constitutes "too long" or "fast enough" will likely vary depending on an individual user's perception as well as the design of the app itself (such as how much information is present in each relevant interface). For the following experiments, this thesis considers latencies below one second to be acceptable; in this range, changes occurring within 100 milliseconds are perceived to be instantaneous, and larger delays are noticeable but well within tolerable times [98] [99]. Note also that a default Android transition lasts for 220 milliseconds [100]—therefore, any interaction occurring within this time will have no perceived delay. Using these metrics, the experiments below attempt to answer the following questions:

- Is the time needed to discover and activate a single AKW acceptable?
- Can an app effectively support multiple AKW opportunities simultaneously?
- Do AKW broadcasts originating from other apps affect overall latency?

6.1.2 SETUP

To perform these experiments, a set of benchmark scenarios were developed using two representative *thing* devices—one a platform for hosting MAAT apps, and the other a physical device that resembles a "real-life" hardware *thing.* A Nexus 9 tablet (2.3 GHz CPU and 2GB RAM) running Android API 23 was used as the MAAT platform, while an Intel Edison development board (500 MHz Atom CPU and 1GB RAM) [95] was chosen as the physical *thing*. These devices were connected to the same WiFi-based private network, with the hardware *thing* running the Atlas architecture and configured to offer a single service described by three keywords. Using this configuration, each device performs no additional tasks—the smart space network activity consists only of AKW interactions and standard metadata tweets.

On the MAAT platform, three simple AKW-enabled apps were developed to evaluate different use cases. Each app presents a unique configuration of AKWs to best fit the goals of a specific scenario, defined as follows:

- The *Simple App* consists of a single AKW (a layout containing a text view and a button) represented by two descriptive keywords, designed to bind successfully with the service offered by the physical *thing*.
- The *List App* uses the same AKW as the Simple App, but instantiates it within a list view. Here, the app displays between 2 and 10 rows of the same AKW layout—when the AKW is bound, all rows update in response.
- The *Complex App* sets up a large number of unique AKW layouts (between 4 and 20 following the format of the Simple App AKW) organized in a grid, each targeting a separate service instance on the physical *thing*.

Each benchmark scenario aims to calculate the overall latency, referred to as *activation time*, under these conditions. In the following subsection, this activation time is broken into three measurements based on the lifecycle of an AKW: 1) *Opportunity Discovery,* a summation of the network transmission times needed to

send an AKW tweet and receive a bind response, 2) *Keyword Match*, the time required by a *thing* to compare an AKW's descriptive keywords with those of its services, and 3) *UI Update*, the period an Android app spends between invalidating an AKW layout and redrawing its interface elements. These values are measured directly (using *System.nanoTime* [101] on Android and *std::chrono::system_clock* [102] on Atlas), except for Opportunity Discovery, which is equivalent to the app's network round trip time for an AKW interaction minus the target's Keyword Match time.

6.1.3 DATA

The first scenario uses the Simple App to calculate the activation time of a single AKW. This aims to determine a lower bound for the latency of a potential MAAT interaction, where each device performs the minimum amount of work necessary to treat an app as a *thing*. Such a lower bound represents the most ideal perceived delay a user may experience. Figure 6-1 shows the average activation time over 100 AKW events: here, Opportunity Discovery is responsible for a majority of the time taken, while in comparison, the Keyword Match and UI Update times are minimal.

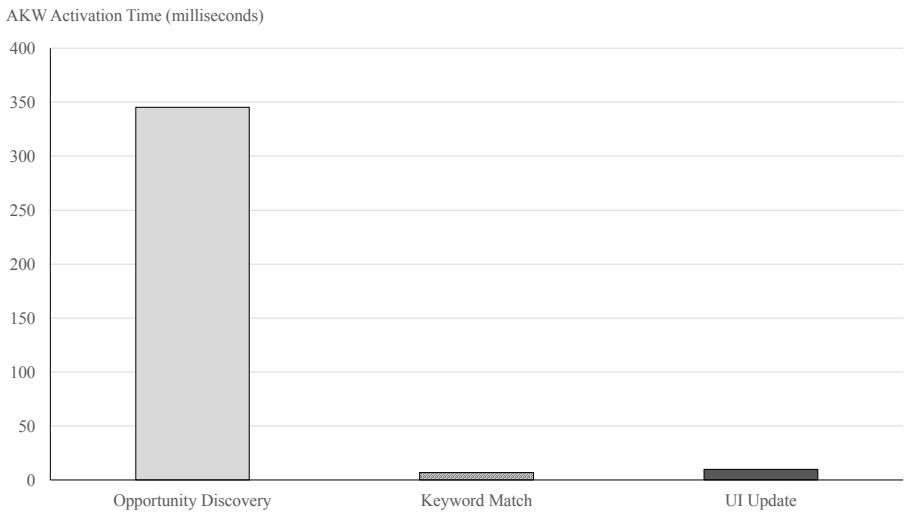

Figure 5-1: Segmented activation time for a single actionable keyword.

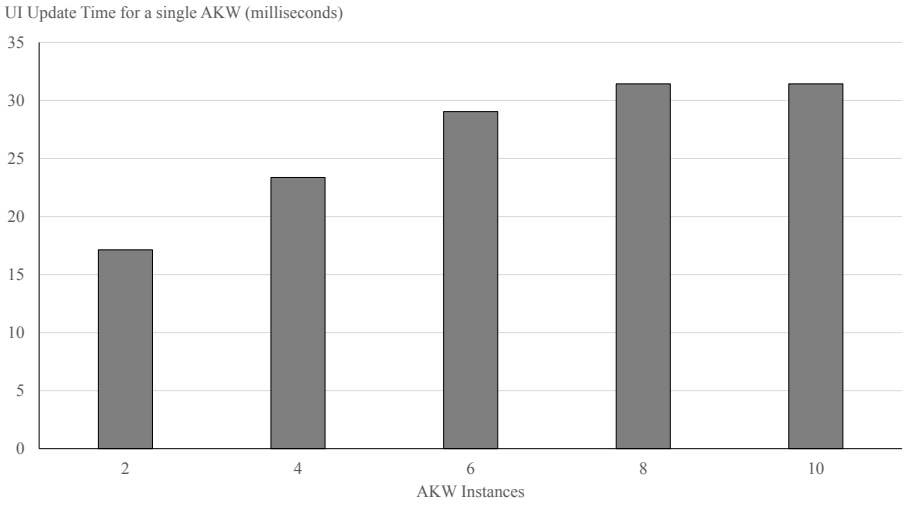

Figure 6-2: UI update time for multiple instances of the same actionable keyword.

The next scenario focuses on the performance of the MAAT Android library; namely, the ActionableLayout logic and behavior. Using the List App, the UI Update time is recorded while varying the number of active list rows. This represents the time needed for a layout to register its binding and to draw any new elements (the buttons). Since a single AKW is always used, only the UI Update time will increase as more instances are shown. Figure 6-2 shows the average update time over 100 interface bindings for each row count—as the number of active layouts increases, so does the time taken, until leveling off at around 8 list rows. This is because rows 8-10 reside outside the list's view bounds and will only update once visible to the user.

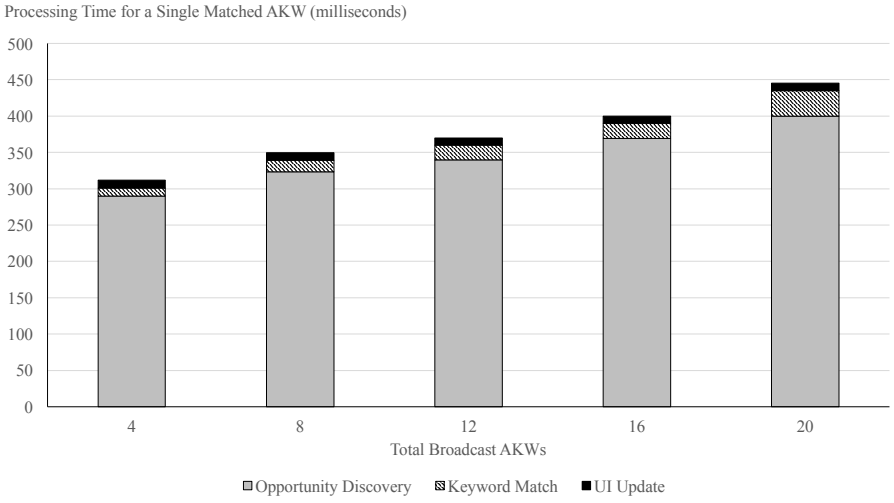

Figure 6-3: Processing multiple actionable keywords with only one match.

The third scenario considers the effect that "background" tweets (that is, other AKW messages that will find no matches within the user's smart space) have on a *thing* device and on activation time. This scenario configures the Complex App such that only one AKW will match on the physical *thing* device, while the others must still be received and compared before being rejected. As with the previous figures, figure 6-3 shows the averages of 100 AKW events for varying numbers of broadcast tweets. The additional network load has a reasonable effect on the *thing* device, increasing Opportunity Discovery times as well as Keyword Match times. These increases are somewhat limited, however, as the Atlas architecture multithreads the receipt and comparison of AKW messages.

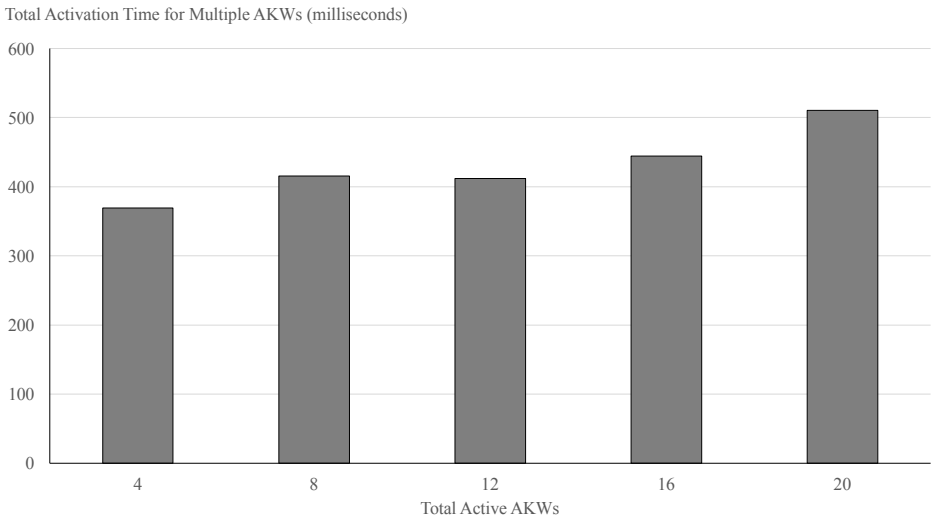

Figure 6-4: Processing multiple actionable keywords where all are matched.

The last scenario aims to test the limits of the actionable keyword concept. Here, the Complex App is configured to match all of its AKWs evenly across four physical *things* (using additional Intel Edison boards), each with up to five services. Figure 6-4 again shows the average measurements over 100 trials for each app configuration. Note that because each physical *thing* receives and processes AKWs independently, only the total activation time of the set is shown; this represents the time between sending the first AKW tweet and updating the final UI element. These measurements

also show a steady increase as more AKWs are activated, with each taking slightly more time than the equivalent configuration in figure 6-3.

6.1.4 Conclusion

Overall, the latency actionable keyword concept remains acceptable (under one second, as defined in subsection 6.1.1) across a variety of situations. The first scenario shows a basic AKW interaction takes about 370 milliseconds at best, or about 150 milliseconds in addition to a regular Android view transition (rather close to being perceived as instantaneous). The networking capabilities of the *thing* devices play a significant role in this benchmark—the speed at which tweets can be exchanged has the greatest effect on perceived latency. The second scenario shows that Android AKW library is also quite performant; changes received from the smart space can be reflected within the interface almost immediately. For example, AKW layout changes propagate in about 2 frames (33 milliseconds) at 60 frames per second, even with many active instances.

This performance persists even under more demanding conditions, such as those described in the remaining scenarios. The third scenario shows that latency remains under 500 milliseconds even when many AKWs are being broadcast to the smart space (such as those from other apps or devices). Here, the *thing* device again plays a large role in the perceived latency, as the Keyword Match time steadily increases in addition to the Opportunity Discovery. The final scenario shows that even in an unlikely scenario (20 AKW layouts filled almost the entirety of the Android tablet's screen), latency remains around 500 milliseconds—while an app may define many keywords, the number visible in a single interface is likely to be less. Since offscreen AKWs have less impact as they update at a slower rate (described in the previous subsection), latency is likely to remain reasonable across most loads.

6.2 Experiment II: MAAT Plugin Evaluation and User Study

In addition to the user-facing performance of the MAAT framework and its interactions with *thing* devices (as evaluated in section 6.1), this thesis also places emphasis on the app developer's role in the creation of mobile app *things.* The Android Studio IDE plugin, described in chapter 5, derives from this emphasis and attempts to simplify the process of adding IoT features to existing mobile app projects. Here, Android developers without extensive IoT experience are the primary target: how useful is such a plugin in this case? This section presents a user study to evaluate how developers use the plugin in a variety of mock IoT scenarios, which are then quantified into a set of usability metrics to describe the utility of the plugin in more objective terms. Finally, a survey is used to assess developers' personal satisfaction with the plugin and the concept overall.

6.2.1 Goals

As described in chapter 4, the MAAT concept places higher expectations on the app developer compared to previous designs; to be effective, individual interface elements must be manually identified as IoT interactions and paired with carefully chosen descriptive keywords. With these more specific requirements, simplifying and integrating the concept into normal app development processes becomes critical when considering ease of use and adoption amongst app developers. The IDE plugin aims to fill this role, prompting the developer for structured information and keywords, which are used to generate appropriate logic and interface code automatically. All of the steps needed to create a new AKW and associated layout are contained within this plugin, such that developer can add MAAT features to an app without missing or forgetting components.

While the plugin sets up this streamlined process, it still requires developers to understand all of the information needed for an AKW interaction—not much can be deduced or inferred by the system. For example, even with the keyword repository detailed in chapter 5, developers must still accurately search for relevant

thing devices when choosing keywords. Considering this, the following experiment presents a set of development tasks involving AKWs and MAAT, along with metrics based on the time taken and number of errors made, to evaluate how effectively developers use the plugin while creating a *thing*-like app. A second part of the experiment complements these tasks, using a survey to quantify developer satisfaction and plugin usability. Overall, this experiment attempts to determine how well the plugin fits into a typical app developer's development process.

6.2.2 SETUP

To perform this experiment, eight participants with varying levels of experience in Android development and Android Studio were recruited. These developers, mainly coworkers and acquaintances, all had at least some experience with developing traditional android apps in a either a personal or professional capacity. Prior to the experiment, each participant was given a short explanation the MAAT framework along with some brief plugin documentation. For the study itself, each participant was asked to complete three timed development tasks with increasing complexities, summarized in table 6-1. The first task provides the developer with a pre-made app and interface, a portion of which must be converted into an AKW layout. The second task requires the developer to create a new interface with two separate AKWs of their choice. Finally, the third task introduces another pre-made app, this time with a list view and data adapter. Here, the developer must integrate an AKW into the list, such that each row contains its own instance. After completing these tasks, each participant was asked to complete an online survey, with metrics ranging from how easily the plugin is used, to how well the plugin fits into an existing workflow.

Table 6-1: MAAT usability study task descriptions.

Task	Description	Expected Duration (minutes)
1	Create a simple AKW	5
2	Create two AKWs in a single interface	10
3	Use list data to create an instanced AKW	15

To evaluate the experience of each participant, the time taken to complete each task, along with any questions or errors encountered, was recorded. These measurements are then converted into three usability metrics derived from the ISO/IEC 9126-4 Usability Standard [103]:

- *Effectiveness* represents the accuracy and completeness with which the developers were able to achieve the specified goals. This calculation is reflected below.

$$Effectiveness = \frac{Number\ of\ tasks\ completed\ successfully}{Total\ number\ of\ tasks\ undertaken}\ x\ 100 \qquad (1)$$

- *Efficiency* describes the resources expended in relation to the effectiveness of the developers. Efficiency is measured in terms of *task time* (the time in minutes a participant takes to complete a task successfully). This can be computed either as the *time-based efficiency* or the *Overall Relative Efficiency* (ORE), both shown below. ORE represents the ratio of the time taken by participants who successfully complete a task to the time taken by all users.

$$Time\ Based\ Efficiency = \frac{\sum_{j=1}^{R}\sum_{i=1}^{N}\frac{n_{ij}}{t_{ij}}}{NR} \quad and \quad ORE = \frac{\sum_{j=1}^{R}\sum_{i=1}^{N}n_{ij}t_{ij}}{\sum_{j=1}^{R}\sum_{i=1}^{N}t_{ij}} \qquad (2)$$

Where:

N	= the total number of tasks
R	= the total number of users
n_{ij}	= the result of task *i* by the user *j*; where success = 1, else = 0
t_{ij}	= the time spent by user *j* to complete task *i* successfully, or until quitting

- *Satisfaction* represents the comfort and acceptability of the plugin. This is measured through a standardized questionnaire administered after the tasks are completed.

6.1.3 DATA

The results of the tasks, measuring effectiveness and efficiency, are shown below in table 6-2. Apart from Task 1, the tasks were completed either within the allocated time or much sooner. The discrepancy with Task 1 can be attributed mainly to the initial unfamiliarity with the MAAT framework in general. While participants understood the initial documentation, some hands-on guidance with the concepts was often needed initially. However, once the developers gained this intuition, few issues were encountered in future use. This is especially prevalent in Task 2, where the concepts from Task 1 could be applied directly. Task 3 was the most complex, introducing a new concept and mixing in some traditional Android development, but overall was completed around the expected time. The participants encountered eight errors total, for an average of one error per three tasks. Most errors in Task 1 resulted from the unfamiliarity described above, while Task 3 errors related to the one-to-many AKW concept and "connecting" this to the list data.

Table 6-2: MAAT usability study task results.

Participant	Task 1 Time (# Errors)	Task 2 Time (# Errors)	Task 3 Time (# Errors)
1	8 minutes (1 error)	4 minutes (0 errors)	10 minutes (0 errors)
2	12 minutes (1 error)	7 minutes (0 errors)	14 minutes (1 error)
3	10 minutes (0 errors)	5 minutes (0 errors)	10 minutes (0 errors)
4	10 minutes (0 errors)	6 minutes (0 errors)	14 minutes (1 error)
5	5 minutes (0 errors)	4 minutes (0 errors)	12 minutes (1 error)
6	10 minutes (1 error)	6 minutes (0 errors)	16 minutes (0 errors)
7	6 minutes (0 errors)	5 minutes (0 errors)	15 minutes (1 error)
8	10 minutes (1 error)	4 minutes (0 errors)	11 minutes (0 errors)

All of the participants were able to complete their tasks successfully; therefore, only the tasks done without error are considered successful when calculating the effectiveness of the MAAT plugin—these results are shown in table 6-3. This same metric is used when calculating the efficiency of the plugin; using the Overall Relative

Efficiency method described above, the efficiency equals (119/214), or 56%. This figure is relatively low due to treating tasks with any error as unsuccessful, especially considering the need for familiarity mentioned previously. For example, if the errors from Task 1 are discounted, the ORE jumps to 74%.

Table 6-3: Calculated effectiveness of the MAAT plugin.

Task	Computation	Effectiveness	Average Effectiveness
1	4/8	50%	
2	8/8	100%	67%
3	4/8	50%	

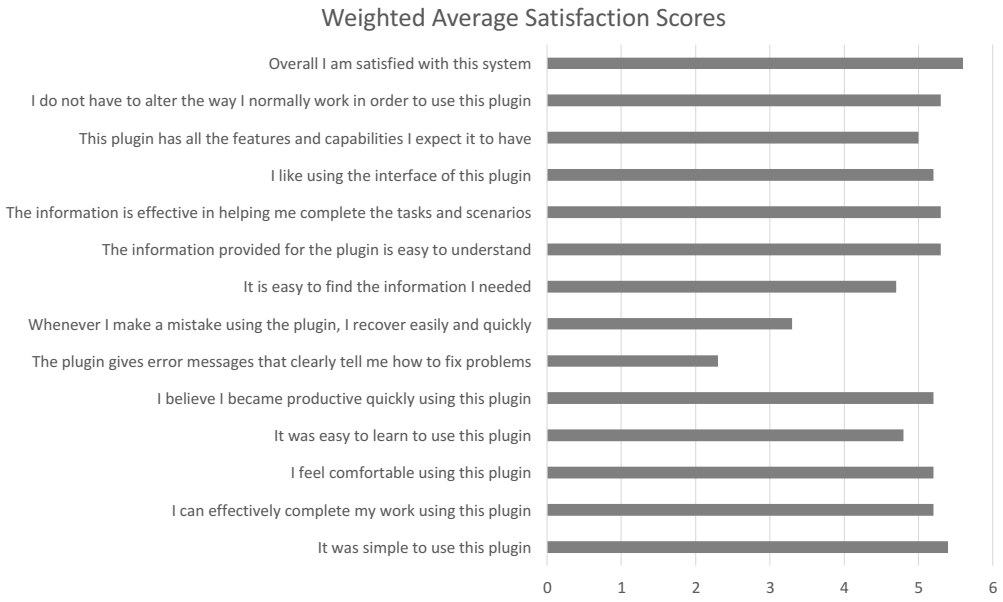

Figure 6-5: MAAT satisfaction survey results.

Finally, figure 6-5 shows the average results of the participants' responses to the 14 questions of the developer satisfaction survey. Almost 88% of the participants felt the plugin was easy to use, and a similar number felt they could complete their work effectively using the plugin. Importantly, most participants believe they would not have to significantly alter the way they worked to use the plugin. Only two questions on the survey received a weighted score less than 4 out of 6; most participants felt

that the plugin did not provide clear error messages or facilitate quick recovery from mistakes. Overall, however, participants indicated they were satisfied with their use of the plugin.

6.2.4 Conclusion

Overall, most participants were able to successfully use the plugin, and reacted positively to the concept as a whole. However, the MAAT library has a noticeable "onboarding" period, highlighted by the times of Task 1. The lack of proper error handling and help messages (reflected in the results of the survey) likely contributed to this; while the plugin was mainly in the prototype stage, features such as proper error messages and in-plugin guidance should be a focus of the experience, and may have limited questions and task errors at this stage. Still, even with these roadblocks, most of the participants were able to get up to speed quickly over the span of three tasks and 30 minutes, and rated the interface and features of the plugin highly.

While the effectiveness and efficiency results are lower in comparison, this can be partially explained by the strict definition of a task error. During the tasks, participants could ask questions that would be marked as an error if the needed guidance was significant enough. Again, this is mainly seen in Task 1, where the most "hands-on" assistance was required. As described in the previous subsection, ignoring these errors results in a positive change to the overall effectiveness and efficiency results. Taking this into consideration, while still requiring some initial level of IoT and framework knowledge, the MAAT plugin appears to generally improve a developer's ability to create new AKW interactions and enable *thing*-like features within their mobile apps.

6.3 Experiment III: Democratization in Mobile Health Apps

Democratization, introduced in chapter 3, refers to a *thing*'s ability to interact outside its vendor-supported software ecosystem. While a variety of open health- and IoT-related standards exist, many devices tend towards their own "siloed"

ecosystem, utilizing vendor-specific interactions and cloud services. This effect is most noticeable in mobile companion apps, where a user with many *thing* devices may begin to experience "app overload" [50] if they buy into a variety of manufacturer ecosystems. While these silos do come with some advantages (especially from the vendor's point of view), how do they compare to an ideal democratized app? This section analyzes the behavior of commercial wellness and personal health IoT devices from a variety of manufacturers, recording a set of performance metrics. Similar measurements are then performed on a custom democratized app with the same capabilities, such that the results can be compared to assess the potential benefits of improved democratization in companion apps.

6.3.1 GOALS

Democratization is one of the four overarching health IoT requirements presented in this thesis. The requirement focuses on enabling flexible interactions across *thing* devices, but is also an important part of an effective companion app: even if a *thing* device supports democratized communication, an app with the same capabilities must exist to take full advantage of its features in a typical health IoT ecosystem. In practice, some vendors may utilize a low level of democratization by supporting their entire device lineup, along with a limited selection of third-party devices, within a single app. However, many follow a one-app-per-*thing* pattern that forces users to maintain a variety of mobile applications; here, a true democratized app has the greatest potential to limit this issue, even across vendors.

Democratized apps may also play a role when considering their resource consumption (such as memory usage, storage space, or background activity) compared to multiple vendor-specific apps. While this is likely not an issue on most modern mobile devices, other metrics such as battery usage may still be at a premium—especially when considering health apps are more likely to be used multiple times per day, or even continuously. In addition to the perceived benefits described in chapter 3, the following experiment attempts to quantify any

performance benefits that can be derived from a democratized mobile app versus multiple siloed apps.

6.3.2 Setup

The following experiment uses a set of commercial health and wellness IoT devices, shown in table 6-4, to serve as the basis for comparison. These devices, each from a different manufacturer and with their own companion app, include a blood pressure cuff, a pulse oximeter, an electrocardiogram (EKG) device, and a sleep tracking mat; together, they cover a variety of health IoT ecosystems, functionalities, and communication methods. To prepare the devices, each was set up as a normal user—charging fully (when applicable), downloading the companion app, and creating an account if required—and paired with the experimental mobile device (a Google Pixel 3a running Android 10).

Table 6-4: A breakdown of a sample of commercial health IoT devices.

Device	Pulse Oximeter	Blood Pressure Cuff	Sleep Tracker	Electrocardiogram
Brand	iHealth Labs [104]	A&D Medical [59]	Withings [105]	AliveCor [58]
Model	Air PO3	UA-651BLE	Sleep Analyzer	KardiaMobile
Connectivity	Bluetooth Low Energy	Bluetooth Low Energy	WiFi	BLE/Microphone
Proper Use	Signal Strength	Cuff Position	N/A	Signal Strength
Device API	Proprietary	PCH Alliance [9]	Proprietary	Proprietary
Mobile App	iHealth MyVitals	A&D Connect	Health Mate	Kardia
Login	Yes	Optional	Yes	Yes
Cloud API	Yes	Partners	Yes	No

To represent an equivalent democratized app, a simple Android application capable of Bluetooth Low Energy (BLE) and WiFi communication was developed. This app is designed to interact with the same commercial devices as above, or with an

analogous simulation for those using proprietary interfaces. In this manner, the app implements the following interactions:

- A blood pressure reading, obtained directly from the A&D Medical blood pressure cuff, as it conforms to the PCH Alliance guidelines [9]—readings are accessible through a standard BLE Health Data Profile (HDP) [11].
- An SPO2 reading, obtained via a simulated HDP "PLX Continuous Measurement," as the iHealth pulse oximeter uses a proprietary BLE profile. This simulated reading models a previous interaction with the real device.
- Sleep data, through a REST service that returns a chunk of simulated data based on the Withings service. The sleep mat is unique in that it passively records data overnight, using the app mainly for review on the next day.
- An EKG reading, through a unique simulation: in addition to BLE, the Kardia device transmits the actual EKG as high-frequency sound decoded by the mobile device's microphone. To simulate at least the collection of this data, microphone activity is recorded during the measurement, which is actually simulated by an HDP heart rate characteristic.

Note that all apps (except for A&D Connect) also require the user to initially log in through their respective cloud services; however, the democratized app does not replicate these features. In both the commercial and democratized cases, each measurement is recorded for 30 seconds; this is the time required for the electrocardiogram and a reasonable average for the blood pressure monitor, while the pulse oximeter can operate continuously for any amount of time. The sleep mat again is an exception, connecting to the REST service only momentarily; however, this still represents a substantial amount of communication. Once measurements are obtained for all four services, performance and battery usage can be measured with the Android Debug Bridge *batterystats* [106] command. This command provides a dump of system events and a breakdown of power usage across the device's hardware and applications, tracked by the Android operating system itself.

6.3.3 DATA

For the commercial apps, each trial measured the power usage of a typical un-democratized "session:" a single use of each device and its companion app, in sequence. Each vendor app was opened, used to perform and review a measurement (taking about 45 seconds total), and immediately closed, confirming any background processes were killed as well. For the democratized case, a similar pattern was followed—the app was left open for 3 minutes while its four measurements were performed in sequence. This ensures a similar amount of screen-on time for both cases, such that the display's power usage does not skew the results. Additionally, the device was fully charged and set to 75% display brightness before each trial.

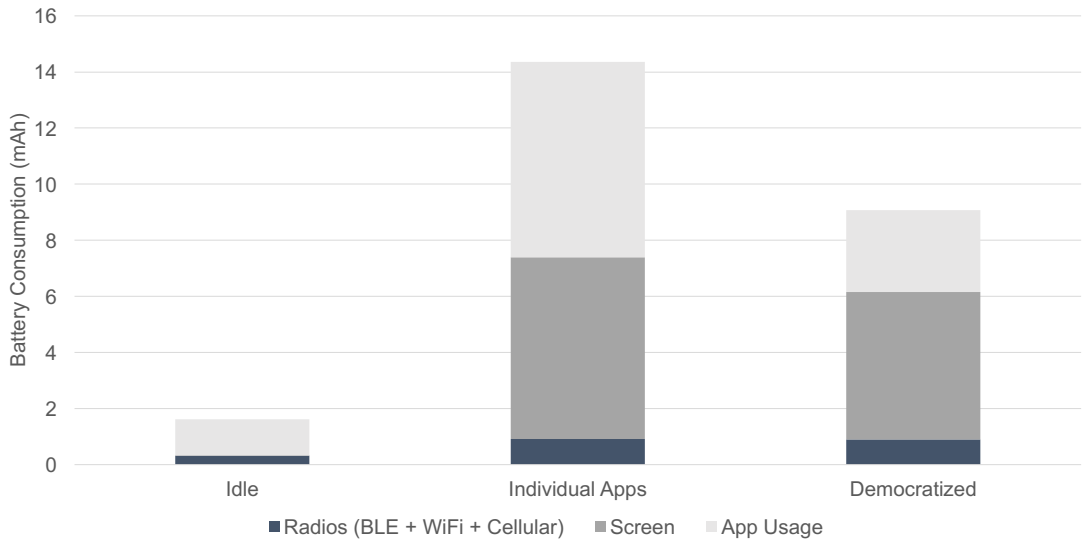

Figure 6-6: Battery consumption in commercial and democratized health apps.

Figure 6-6 shows the average results of 10 trials for each case, as well as the base "idle" consumption of the mobile device (screen off). These results show that under the same measurement conditions, the individual commercial apps consume about 50% more energy than the single democratized app. In both cases, the energy usage of the communication hardware and screen is similar: most difference comes from the software itself. To further investigate, additional trials were performed using

each commercial app individually over 3 minutes. Figure 6-7 again shows the results across 10 trials—the energy consumption of each app, while similar, shows a correlation with the amount of additional network traffic performed. Therefore, the lack of cloud communication features must also play a role in the energy savings of the democratized app.

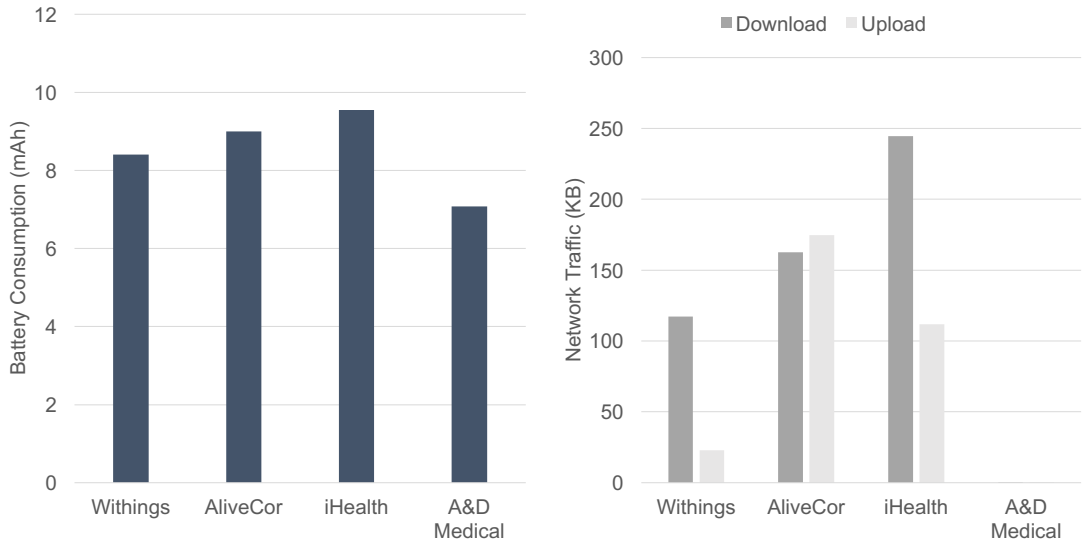

Figure 6-7: Battery consumption and network traffic in individual commercial apps.

6.3.4 Conclusion

Overall, the democratized app exhibits a reasonable decrease in energy usage compared to the equivalent set of commercial apps. Much of the individual apps' additional usage is likely due to the overhead of loading and switching between the four applications, whereas the democratized app only needs to be loaded once. Networking and cloud requirements also play a part, especially when interacting with three separate services in a single session. This is particularly evident when comparing the A&D Medical app in the second part of the experiment—without any user accounts or cloud services, the app consumes the least energy of the four.

While these differences are relatively small (about 5mAh between the two scenarios), these devices are meant to be used multiple times throughout the day (or

even continuously in some cases), where such numbers can begin to add up. For example, a user may take a set of readings five times a day; assuming the user is not perfectly efficient and takes an additional 2-3 minutes per use, this process would consume about 5% of the Pixel 3a's 3000mAh battery. In another example, a user could be monitoring for potential COVID symptoms throughout the night using a continuous ring oximeter (such as Wellue O2Ring [107]) that repeats measurements automatically. Additionally, considering users with older phones containing batteries that are smaller or more worn out (unable to hold a full charge), this could easily account for over 10% of their daily use. Therefore, even with this relatively small difference, democratized apps can help limit any future impact as health IoT *things* become more common.

6.4 Experiment IV: Overhead of Health Device Requirements

While the requirements introduced in chapter 3 present a variety of features with potential to increase the overall utility of a health IoT *thing*, many also come with new needs or expectations from the devices. For example, such expectations may come in the form of additional resources, communication capabilities, or software logic. While many of these are likely to be easily supported by the architecture, they still come with their own impacts and overheads. Especially when considering the variety of low-power *things* within a health IoT ecosystem, how does the overhead of these requirements compare to their presented advantages? This section presents and evaluates minimal implementations of three requirements—safe use, user identity, and IoTility (while the democratization element is evaluated in section 6.3)—and benchmarks them against a basic health IoT device without these features. The impact of these performance measures can then be weighed against the potential benefits of the requirements.

6.4.1 GOALS

As mentioned above, each of the architectural requirements introduce some form of additional behavior. For example, safe use requires extra logic when sensing and actuating, or even additional hardware components to detect improper use. User identity must manage multiple sets of data through software logic or storage space. IoTility, focusing on interactions between *things*, requires a variety of specialized networking and communication features from the architecture. Additionally, these requirements are likely to have varied impacts, depending on the needs and goals of the specific health device. With this in mind, the following experiments attempt to quantify the base performance overhead of the requirements, using a representative low-power health *thing* to benchmark an implementation of each.

When considering constrained devices, power usage is a primary concern in terms of performance metrics. This is especially true when considering the overhead of IoTility features, as network radios are often a major component of energy consumption. To collect these metrics, the experiments of this section consider each requirement in isolation, layering a representative implementation on top of a simple health IoT system (the low-power device described above, simulating a basic measurement service). This allows the impact of each to be measured without involving the other features of a complete health IoT architecture in the results.

6.4.2 SETUP

The following experiments use a Nordic nRF52840 [108] ARM-Cortex microcontroller dongle to represent the target low-power health *thing*. This family of devices is used in various smart watches and fitness bands, and offers connectivity and power-saving features that are common amongst similar systems-on-a-chip used in commercial health IoT devices. The device was programmed to act as a basic BLE continuous-monitoring health *thing*, which performs a simulated reading at the rate of twice per second. Each of these readings is collected with minimal processing or validation before being sent out—this represents a device with minimal IoT architecture and

requirement support, and is used to record the baseline power usage. Each of the following representative requirements was then integrated into separate instances of the device firmware:

- *Safe Use:* when considering safe use with continuous measurements, the reliability of a reading may change in real-time as the user shifts and moves. To ensure the accuracy of the measurement overall, the device may share additional parameters as feedback. For example, an EKG device may report the amount of skin contact on the electrodes, while a pulse oximeter may detect when the finger is not inserted far enough. To represent this scenario, algorithm 1 is used to simulate the collection of "proper use metrics" from the device's hardware inputs (line 3). These additional values must be converted to an accuracy or validity measurement (line 4) by applying comparisons or calculations. These measurements are different from the actual readings offered by a device (obtained through *get_measurement()* in line 5). Finally, the proper use metrics can be packaged and displayed alongside the actual reading (line 5) or transmitted (line 6) to a companion app for further handling. In this experiment, and analog GPIO reading represent the safe use metric.

Algorithm 1: Representative Safe Use

```
1 Function update
2     while connected do
3         proper_use_metric = read_additional_sensors();
4         accuracy = calculate_accuracy(proper_use_metric);
5         service_values = Pair(get_measurement(), accuracy);
6         update_characteristic(service_values);
7         sleep(0.5 seconds);
8     end
9 end
```

Figure 6-8: The representative safe use algorithm.

- *User Identity:* similar considerations must be made for the user identity requirement: before measurements begin, a device may need to be informed

about the active user. For example, a blood pressure monitor may accept a profile handle to store the user's measurement history, while a body weight scale may request the user's height to calculate BMI. Algorithm 2 implements an additional BLE service characteristic, allowing the user to provide additional information before the reading takes place (line 2). This may be used to verify the user, or to augment and associate the reading with more details (line 3). Once an identity is verified (likely for a limited time), subsequent device measurements (line 5) are "stamped" with the verified user info before transmission (lines 6 and 7).

Algorithm 2: Representative User Identity

```
1  Function update
2      profile = wait_for_characteristic_write();
3      verify_user_profile(profile);
4      while connected do
5          measurement = get_measurement();
6          user_data_record = prepare_value(measurement, profile);
7          update_characteristic(user_data_record);
8          sleep(0.5 seconds);
9      end
10 end
```

Figure 6-9: The representative user identity algorithm.

- *IoTility:* device IoTility, on the other hand, is a much more demanding requirement in the context of a low-power Bluetooth device. For this case, a more blue-sky is adopted, where the device periodically searches for other BLE services it can use in addition to its normal capabilities. For example, instead of asking for identity parameters as in algorithm 2, the device can independently search for services offering these values. This scenario is represented in algorithm 3: the device initially acts as a BLE central, scanning for related services (line 2) and pulling identity information from any matches (line 3 and 4). Device readings can then be shared normally as a BLE peripheral (lines 5 and 6). This represents a simple IoTility scenario compared to those offered in chapter 3.

Algorithm 3: Potential Device IoTility

```
1 Function update
2     services = scan_for_peripherals();
3     if user_profile_service in services then
4         profile = request_user_profile(user_profile_service);
5         while connected do
              /* Collect measurements as in algorithm 2 */
6         end
7     else
          /* Repeat scan */
8     end
9 end
```

Figure 6-10: A potential device IoTility interaction.

Finally, energy measurements for each experimental scenario are taken by powering the device with an external 5V source in series with a low-value shunt resistor. The voltage drop across this resistor—converted into a current measurement using Ohm's Law—can then be used to calculate the power usage of the device over a one minute period. Note that unlike a real device, there is no additional load from supporting circuitry or measurement devices; the microcontroller is used with only a single on-board LED.

6.4.3 DATA

The results of these simulations over 10 trials are shown in figure 6-11. In this figure, each scenario is represented in two "states:" one where the device is actively advertising or scanning for the entire duration, and one where a discovery and connection occurs immediately. In all cases, the power consumption is less once a connection is established due to the properties of the BLE radio; broadcasting to the smart space requires more energy than sustaining a single connection. This difference is most significant in the IoTility scenario, which performs additional advertising procedures.

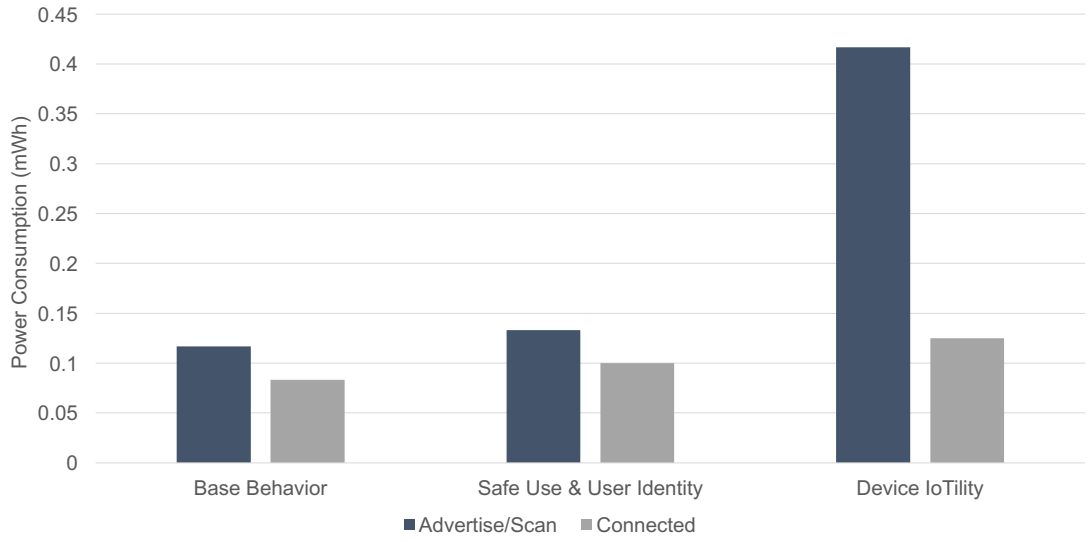

Figure 6-11: Power consumption in the different device requirement scenarios.

Compared to the baseline, the impact of the safe use and user identity requirements is minimal in their respective scenarios. Together, these requirements represent potential for increased usage in hardware, processing, and networking; however, within the experiments, the increases due to additional service writes and analog-to-digital conversions are only a small contribution. On the other hand, the impact of the IoTility scenario is more significant; the power cost of performing a BLE scan is greater in comparison to the baseline measurements. However, once a valid service is discovered, power usage decreases to better match the other scenarios.

6.4.4 Conclusion

Overall, these results show that the potential costs of the safe use and user identity requirements can be acceptable in low-power health devices. In fact, some of these features can already be found in some commercial devices (as noted in section 6.3). Still, this impact may be greater in real-life scenarios, depending on the components required to meet the goals of the *thing*. For example, a device using complicated calculations or additional power-hungry hardware (while monitoring proper use, for example) would experience a greater impact to power usage. Overall, however, the

minimal impact of these experimental scenarios leaves ample room for any additional overhead.

In comparison, device IoTility has a relatively high cost on such devices; searching for potential interactions requires significant energy unless performed sparingly (the experimental scenario uses a scanning duty cycle of 50%). In this case, inter-*thing* communications may not be feasible—beyond direct interaction with a paired controller or companion app—in low-power health devices. However, such devices may still support IoTility through other means, such as the proxy interfaces described in chapter 3. Alongside these low-power devices, more capable health *things* (such as mobile phones) are also common, and capable of "picking up the slack" with high-overhead features.

6.5 Summary

This chapter presented a set of experimental evaluations designed to evaluate the architectural implementations of the overarching requirements, as well as the Mobile Apps As Things concept and actionable keyword programming enabler. One experiment considered the energy impact of the Democratization requirement in the context of companion mobile apps, comparing an idealized shared app against a set of individual commercial applications. While power consumption is low in both cases, the democratized app showed reduced energy usage and network activity for an equivalent workload. Another experiment investigated the performance impact of supporting the Safe Use, User Identity, and IoTility requirements on constrained hardware. While the representative Safe Use and User Identity behaviors had little impact on the power consumption of the device, that of the IoTility behavior was more significant; constrained devices would likely need other devices (such as smart phones) to assist in the support of this requirement.

For MAAT, this chapter considered the time latency involved in a complete AKW interaction: that is, the time between broadcasting keywords from the app and the new user interface element appearing. In all cases, involving a varying number of

devices and active AKWs, this latency remained within defined reaction times. These results showed that the user will be presented with new interactions quickly, reducing the potential for confusion or missed opportunities. Within MAAT, the IDE plugin was also evaluated through a small user study with Android developers. After creating a set of sample applications, each developer answered a questionnaire and was evaluated based on a set of success- and time-based metrics. Reception to the plugin was generally positive, although efficiency and effectiveness varied depending on the specific task.

The experiments in this chapter evaluate the overall impact that the requirements presented in chapter 3 have on a representative Atlas Health IoT device, along with some key ecosystem features that integrate closely with the *thing* architecture. These features and requirements have the potential to play a core role in a variety of health IoT scenarios and applications, bringing significance to their performance and usability aspects alongside the architecture itself. Still, many of the individual architectural features introduced in chapter 3 are open to additional focus and experimentation; this potential is discussed in the future work of chapter 7.

CHAPTER 7: CONCLUSION

Health IoT is preparing to be a prominent force in the future landscape of personal health and wellness. Traditional medical devices with smart and connected features are opening up new opportunities in disease prevention and treatment, and empowering users outside of traditional healthcare settings. However, treating these devices as traditional IoT *things* is unlikely to utilize their full potential; a variety of healthcare scenarios introduce functional requirements that are unique or less of a focus in other IoT domains. To fully achieve the vision of health IoT and its role in future healthcare transformations, the influence of these requirements must be carefully considered from the perspective of stakeholders (including end users, developers, vendors, and healthcare providers), *thing* devices and services, and the greater healthcare ecosystem.

7.1 THESIS REVIEW

This thesis focused on the role of a *thing architecture*—the firmware running on and interacting with these connected health devices—in this health IoT vision: what features and capabilities must a health IoT device provide to best support the specialized needs of personal health and healthcare delivery integration scenarios? To answer this research question (RQ2), alongside the other research questions presented in chapter 1, the thesis presented a set of *requirements*, or overarching features and concepts that reflect needs and priorities common amongst health IoT systems (answering RQ1). These requirements were considered from the perspective of a device's software functionality, and then used to identify features that could contribute to the creation of effective health IoT applications and scenarios. Once identified, the thesis further investigated a selection of these requirement features, implementing and integrating them into a modern *thing* architecture and ecosystem (answering RQ3). Finally, these features were evaluated through a set of experiments to determine performance characteristics and other metrics (also answering RQ3)

that are likely to be relevant to future stakeholder groups (based on section 3.1.2's discussion of RQ4).

Four overarching requirements were identified in this thesis: democratization, safe use, user identity and privacy, and a new concept called *IoTility*, describing a *thing's* ability to enable safe and meaningful interactions with other *things,* devices, and users. The thesis focused particularly on democratization and IoTility, identifying a set of software features that could contribute towards meeting these requirements (RQ2). For example, shared APIs, flexible user interfaces, and data channeling were identified as important features within democratization, while IoTility introduced concepts on *thing*-like mobile apps, proxy interfaces, and incentivization (among others) as key components.

Additionally, some of these features were integrated into the Atlas Thing Architecture—forming the Atlas Health IoT Architecture—to investigate a device architecture's role in a health-ready IoT ecosystem (RQ3). In addition to the architecture itself, the concept of Mobile Apps As Things (MAAT), along with the *actionable keywords* programming enabler, was also introduced to better integrate and enable the capabilities of *companion mobile apps.* These apps, which control the behavior of a *thing* and interface it with the cloud, are ubiquitous in the context of present-day connected health devices. However, most of these apps are quite limited in the scope of the devices they support, and rarely allow for additional integration. The MAAT concept introduced in this thesis allows app developers (one group of important health IoT stakeholders) to easily integrate similar features and *thing*-like behaviors without requiring specific knowledge of the target devices.

The thesis then described the implementation of these components in detail. This includes internal runtime features such as inter-device communication and a dynamic service model, the complete MAAT ecosystem, including an Android support library and IDE plugin, and modifications to the IoT Device Description Language (IoT-DDL). A new web-based tool was also introduced, allowing developers, vendors, and users (three main health IoT stakeholders) to visually create IoT-DDL files without requiring knowledge of the schema or its implementation. The self-

documenting effect of the presented GUI-based tools (both the IoT-DDL Builder and the MAAT plugin) aims to simplify and encourage use amongst the stakeholders that can enable these new requirements (RQ4).

Next, this thesis presented a set of studies and experiments to evaluate the concepts described above. One experiment evaluated the performance characteristics of actionable keywords and the MAAT concept, investigating how an end user may perceive the new UI functionalities and their overhead. Another experiment introduced a group of Android developers to the MAAT IDE plugin, evaluating its effectiveness and providing insight on its ease of use in creating new mobile apps with advanced *thing*-like functionality. Across all cases, the performance of MAAT remained promising, with latencies remaining well within the bounds of target reaction times, even with many active actionable keywords. Similarly, reception of the IDE plugin was quite positive amongst the participant developers.

Given the prevalence of smartphones and companion mobile apps in the health IoT domain, MAAT represents an important component of the IoTility requirement while also showing some advantages of improved democratization. Mobile app developers can integrate new *thing* capabilities without programming around a specific device or ecosystem, potentially allowing vendors to design device interactions targeting specific apps (rather than the other way around), improving compatibility and flexibility between stakeholders.

In a broader experimental scope, an additional evaluation aimed to determine the general feasibility of the four proposed requirements, in terms of performance (RQ3). This involved comparing the performance characteristics of commercial health IoT companion apps with a representative democratized app, as well as those of *thing* devices running structured implementations of the privacy, safe use, and IoTility requirements. Within a mobile device, the democratized app revealed potential for decreased power consumption compared to similar use of multiple, device-specific apps. On the other hand, while the safe use and privacy cases had limited influence on performance, the IoTility scenario displayed a greater

impact, dependent on the capabilities of the *thing* device and how it communicates with the smart space.

These experiments reveal that democratization can also play a role in device performance, in addition to the perceived end user and ecosystem benefits. Such benefits provide additional reasoning for increased democratization, and may help further encourage these patterns amongst vendors. On the other hand, the impact of the safe use, privacy, and IoTility requirements is likely to be specific to individual *thing* scenarios, especially on more resource constrained devices. However, with increasingly powerful devices—such as smartphones—acting as the "center" of many interactions, IoTility can also be achieved through effective integration with the smart space and other *thing* devices. Overall, these considerations are likely to be critical to vendors and developers attempting to meet the specialized needs of health IoT through these requirements.

7.2 Limitations and Future Work

Overall, this thesis presents a set of requirements that are believed to have a significant impact on a *thing* architecture's potential within health IoT. While these requirements are argued to be an important base, one must again note the potential for additional requirements; the thesis does not attempt to provide an all-encompassing set. For example, specialized health scenarios may reveal the need for additional or modified requirements alongside those presented previously; such a possibility opens up one avenue for future work, especially when considering accessibility and other scenarios where the end user's needs may differ significantly. Similarly, when reflecting on the health device "tiers" presented in chapter 3, this thesis focuses on the needs of personal health devices: while the other tiers may utilize similar features, they may also necessitate additional requirements (especially when considering medical devices and their use cases).

Additionally, the implementation does not provide the full scope of features described in chapter 3. Although all of the requirements are considered in the design

of the architecture, the final result focuses mainly on features relating to Democratization and Device IoTility, while providing a more general framework for other requirements such as Safe Use. Future work in this area may continue to build on the Atlas Health IoT architecture, introducing more concrete implementations of features such as User Identity and Privacy or Incentivization. Similar consideration extends to the evaluation, which is split between requirement performance metrics and an investigation into Mobile Apps As Things. New or existing requirements may be further validated through future work involving limited user studies with stakeholders such as patients and healthcare providers; such user-facing needs may not map directly to the requirements of a *thing* architecture like the ones presented in this thesis but would still provide insight into the needs and future of health IoT.

REFERENCES

[1] Deloitte Insights, "Forces of Change: The Future of Health," The Deloitte Center for Health Solutions, 2019.

[2] GoInvo LLC, "Determinants of Health," 2016. [Online]. Available: https://www.goinvo.com/features/determinants-of-health/. [Accessed November 2020].

[3] iHealth Labs, "About iHealth," [Online]. Available: https://ihealthlabs.com/pages/about-us. [Accessed November 2020].

[4] "Withings," [Online]. Available: https://www.withings.com/uk/en/. [Accessed November 2020].

[5] Libelium, "MySignals," [Online]. Available: http://www.my-signals.com/. [Accessed November 2020].

[6] Apple, "Apple Health," [Online]. Available: https://www.apple.com/ios/health/. [Accessed July 2020].

[7] Google, "Google Fit," [Online]. Available: https://www.google.com/fit/. [Accessed July 2020].

[8] R. Carroll, R. Cnossen, M. Schnell and D. Simons, "Continua: An Interoperable Personal Healthcare Ecosystem," *IEEE Pervasive Computing,* vol. 6, no. 4, pp. 90-94, 2007.

[9] Personal Connected Health Alliance, "Interoperability Design Guidelines for Personal Connected Health Systems," 2019. [Online]. Available: https://members.pchalliance.org/document/dl/2148. [Accessed July 2020].

[10] IEEE Health Informatics, "IEEE 11073-20601-2019 Application Profile: Optimized Exchange Protocol," [Online]. Available: https://standards.ieee.org/standard/11073-20601-2019.html. [Accessed November 2020].

[11] Bluetooth Technology, "Health Device Profile Data Exchange Specifications and Specializations," [Online]. Available: https://www.bluetooth.com/specifications/ assigned- numbers/health-device- profile/. [Accessed November 2020].

[12] "A&D Medical," [Online]. Available: https://medical.andonline.com/home. [Accessed November 2020].

[13] HL7, "FHIR Standard Release 4," 2019. [Online]. Available: https://hl7.org/FHIR/. [Accessed November 2020].

[14] MIT, "Solid," 2017. [Online]. Available: solid.mit.edu. [Accessed November 2020].

[15] "MyData," 2018. [Online]. Available: mydata.org/mydata-101. [Accessed November 2020].

[16] S. Harous, M. El Menshawy, M. A. Serhani and A. Benharref, "Mobile health architecture for obesity management using sensory and social data," *Informatics in Medicine Unlocked,* vol. 10, pp. 27-44, 2018.

[17] S. Deshkar, R. A. Thanseeh and V. Menon, "A Review on IoT based m-Health Systems for Diabetes," *International Journal of Computer Science and Telecommunications,* vol. 8, no. 1, 2017.

[18] O. Debauche, S. Mahmoudi, P. Manneback and A. Assila, "Fog IoT for Health: A new Architecture for Patients and Elderly Monitoring," *Procedia Computer Science,* vol. 160, pp. 289-297, 2019.

[19] R. Lomotey, J. Pry and S. Sumanth, "Wearable IoT data stream traceability in a distributed health information system," *Pervasive Mobile Computing,* vol. 40, pp. 692-707, 2017.

[20] P. Mahalle, P. Thakre, N. Prasad and R. Prasad, "A Fuzzy Approach to Trust Based Access Control in Internet of Things," *Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace and Electronic Systems,* 2013.

[21] J. O'Donoghue and J. Herbert, "Data Management within mHealth Environments: Patient Sensors, Mobile Devices, and Databases," *Journal of Data and Information Quality,* vol. 4, no. 1, 2012.

[22] P. Yang, D. Stankevicius, V. Marozas, Z. Deng, E. Lui and A. Lukosevicius, "Lifelogging Data Validation Model for Internet of Things Enabled Personalized Healthcare," *IEEE Transactions on Systems, Man, and Cybernetics: Systems,* vol. 48, no. 1, pp. 50-64, 2018.

[23] P. Laplante, M. Kassab, N. Laplante and J. Voas, "Building Caring Healthcare Systems in the Internet of Things," *IEEE Systems Journal,* vol. 12, no. 3, 2017.

[24] B. Farahani, F. Firouzi, V. Chang, M. Badaroglu, N. Constant and K. Mankodiya, "Towards fog-driven IoT eHealth: Promises and challenges of IoT in medicine and healthcare," *Future Generation Computer Systems,* vol. 78, pp. 659-676, 2018.

[25] A. Plaza, J. Diaz and J. Perez, "Software architectures for health care cyber-physical systems: A systematic literature review," *Journal of Software: Evolution and Process,* vol. 30, no. 1, 2018.

[26] Arm Limited, "Mbed," [Online]. Available: https://os.mbed.com/. [Accessed October 2020].

[27] J. Voas, "Network of 'Things'," *NIST Special Publication,* 2016.

[28] C. Perera, P. P. Jayaraman, A. Zaslavsky, P. Christen and D. Georgakopoulos, "MOSDEN: An Internet of Things Middleware for Resource Constrained Mobile Devices," *47th Hawaii Int'l Conf. on System Sciences,* pp. 1053-1062, 2014.

[29] "IFTTT," 2019. [Online]. Available: https://ifttt.com.

[30] B. Ur, M. Ho, S. Brawner, J. Lee, S. Mennicken and N. Picard, "Trigger-Action Programming in the Wild: An Analysis of 200,000 IFTTT Recipes," *CHI,* pp. 3227-3231, 2016.

[31] J. Yun, I.-Y. Ahn, S.-C. Choi and J. Kim, "TTEO (Things Talk to Each Other): Programming smart spaces based on IoT systems," *Sensors 16,* vol. 4, p. 467, 2016.

[32] L. Atzori, A. Iera, G. Morabito and M. Nitti, "The Social Internet of Things (SIoT) – When social networks meet the Internet of Things: Concept, architecture and network characterization," *Computer Networks,* vol. 56, no. 16, 2012.

[33] R. Girau, M. Nitti and L. Atzori, "Implementation of an Experimental Platform for the Social Internet of Things," *7th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing,* pp. 500-505, 2013.

[34] W3C, "Web of Things," [Online]. Available: https://www.w3.org/WoT/. [Accessed November 2020].

[35] W3C, "Web of Things Thing Description Proposal," 2020. [Online]. Available: https://www.w3.org/TR/2020/PR-wot-thing-description-20200130/. [Accessed November 2020].

[36] K.-H. Le, S. K. Datta, C. Bonnet and F. Hamon, "WoT-AD: A Descriptive Language for Group of Things in Massive IoT," *IEEE 5th World Forum on Internet of Things,* 2019.

[37] C. Chen and S. Helal, "Sifting Through the Jungle of Sensor Standards," *IEEE Pervasive Computing,* vol. 7, no. 4, pp. 84-88, 2008.

[38] C. Chen and A. Helal, "Device Integration in SODA Using the Device Description Language," in *2009 Ninth Annual International Symposium on Applications and the Internet*, 2009.

[39] A. Khaled, Architecture and Programming Model for the Social Internet of Things, PhD Thesis, 2018.

[40] A. Khaled, A. Helal, W. Lindquist and C. Lee, "IoT-DDL – Device Description Language for the “T” in IoT," *IEEE Access,* vol. 6, pp. 24048-24063, 2018.

[41] J. King, R. Bose, H.-I. Yang, S. Pickles and A. Helal, "Atlas: A service-oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces," in *31st IEEE Conference on Local Computer Networks*, 2014.

[42] S. de Deugd, R. Carroll, K. Kelly, B. Millett and J. Ricker, "SODA: Service Oriented Device Architecture," *IEEE Pervasive Computing,* vol. 5, no. 3, 2006.

[43] A. Khaled and S. Helal, "Interoperable communication framework for bridging RESTful and topic-based communication in IoT," *Future Generation Computer Systems,* vol. 92, no. 1, 2018.

[44] A. Khaled, W. Lindquist and A. Helal, "Service-Relationship Programming Framework for the Social IoT," *Open Journal of Internet of Things,* vol. 4, pp. 35-53, 2018.

[45] I. Azimi, A. Anzanpour, A. Rahmani, T. Pahikkala, M. Levorato and P. Liljeberg, "HiCH: Hierarchical Fog-assisted Computing Architecture for Healthcare IoT," *ACM Transactions on Embedded Computing Systems,* 2017.

[46] L. Catarinucci, D. d. Donno, L. Mainetti, L. Palano, L. Patrono and M. Stefanizzi, "An IoT-Aware Architecture for Smart Healthcare Systems," *IEEE Internet of Things Journal,* vol. 2, no. 6, 2015.

[47] U.S. Food and Drug Administration, "General Wellness: Policy for Low Risk Devices," 2016. [Online]. Available: https://www.fda.gov/media/90652/download. [Accessed October 2020].

[48] U.S. Department of Health & Human Services, "HIPAA Privacy," [Online]. Available: https://www.hhs.gov/hipaa/index.html. [Accessed October 2020].

[49] European Parliament and Council of European Union, "Regulation (EU) 2016/679," 2016. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679&from=EN. [Accessed October 2020].

[50] L. van Velsen, D. Beaujean and J. Gemert-Pij, "Why mobile health app overload drives us crazy, and how to restore the sanity," *BMC medical informatics and decision making,* vol. 13, no. 23, 2013.

[51] Withings, "Smart Temporal Thermometer," [Online]. Available: https://www.withings.com/nl/en/thermo. [Accessed July 2020].

[52] iHealth, "Gluco+ Smart Glucometer," [Online]. Available: https://ihealthlabs.eu/en/64-glucometre-connecte-ihealth-gluco.html. [Accessed October 2020].

[53] D. Dimitrov, "Medical Internet of Things and Big Data in Healthcare," *Healthcare Inform. Res.,* vol. 22, no. 3, pp. 156-163, 2016.

[54] Q. Limbourg and J. Vanderdonckt, "UsiXML: A User Interface Description Language Supporting Multiple Levels of Independence," *Engineering Advanced Web Applications,* p. 325–338, 2004.

[55] A. Ribeiro and A. Silva, "XIS-mobile: a DSL for mobile applications," *ACM Symposium on Applied Computing,* 2014.

[56] Apple, "HealthKit," [Online]. Available: https://developer.apple.com/documentation/healthkit. [Accessed October 2020].

[57] HL7, "FHIR Standard Release 4," 2019. [Online]. Available: https://hl7.org/FHIR/. [Accessed July 2020].

[58] AliveCor, "KardiaMobile," [Online]. Available: https://www.alivecor.com/kardiamobile. [Accessed July 2020].

[59] A&D Medical, "UB-543 Automatic Wrist Blood Pressure Monitor," [Online]. Available: https://medical.andprecision.com/product/ub-543-automatic-wrist-blood- pressure- monitor/. [Accessed July 2020].

[60] X. Kang, L. Zhao, F. Leung, H. Luo, L. Wang and J. Wu, "Delivery of Instructions via Mobile Social Media App Increases Quality of Bowel Preparation," *Clinical Gastroenterology and Hepatology,* vol. 14, no. 10, 2015.

[61] S. A. Weinzimer, G. M. Steil, K. L. Swan, J. Dziura, N. Kurtz and W. V. Tamborlane, "Fully Automated Closed-Loop Insulin Delivery Versus Semiautomated Hybrid Control in Pediatric Patients with Type 1 Diabetes Using an Artificial Pancreas," *Diabetes Care,* vol. 31, no. 5, pp. 934-939, 2008.

[62] R. Bergenstal, S. Garg and S. Weinzimer, "Safety of a Hybrid Closed-Loop Insulin Delivery System in Patients with Type 1 Diabetes," *JAMA,* vol. 316, no. 13, pp. 1407-1408, 2016.

[63] Medtronic Diabetes, "Minimed 780G System," [Online]. Available: https://www.medtronic-diabetes.co.uk/insulin-pump-therapy/minimed-780g-system. [Accessed October 2020].

[64] W. Lindquist, A. Khaled and S. Helal, "IoT-DDL: Device Description Language for a Programmable IoT," *International Conference on Smart and Sustainable Technologies.,* 2020.

[65] W. Dunn, "Designing safety-critical computer systems," *Computer,* vol. 36, no. 11, pp. 40-46, 2003.

[66] E. Thomaz, I. Essa and G. Abowd, "A Practical Approach for Recognizing Eating Moments with Wrist-Mounted Inertial Sensing," *ACM International Joint Conference on Pervasive and Ubiquitous Computing,* pp. 1029-1040, 2015.

[67] IQVIA, "The Growing Value of Digital Health," 2017. [Online]. Available: www.iqvia.com/institute/reports/the-growing-value-of-digital-health. [Accessed July 2019].

[68] Research2Guidance, "mHealth App Economics 2017," 2017. [Online]. Available: research2guidance.com/product/mhealth-economics-2017-current-status-and-future-trends-in-mobile-health. [Accessed 2019 July].

[69] B. Jimmy and J. Jose, "Patient Medication Adherence: Measures in Daily Practice," *Oman Medical Journal,* vol. 26, no. 5, pp. 155-159, 2011.

[70] N. Small, P. Bower, C. Chew-Graham, D. Whalley and J. Protheroe, "Patient empowerment in long-term conditions: Development and preliminary testing of a new measure," *BMC Health Services Research,* vol. 13, no. 7, p. 263, 2013.

[71] D. Lee, A. Helal, Y. Sung and S. Anton, "ituation-Based Assess Tree for User Behavior Assessment in Persuasive Telehealth," *IEEE Transactions on Human-Machine Systems,* vol. 45, no. 10, pp. 624-634, 2015.

[72] D. Lee, S. Helal, S. Anton, S. de Deugd and A. Smith, "Participatory and Persuasive Telehealth," *Gerontology,* vol. 58, pp. 269-281, 2011.

[73] C. Jaffe, C. Mata and S. Kamvar, "Motivating Urban Cycling through a Blockchain-Based Financial Incentives System," *ACM International Joint Conference on Pervasive and Ubiquitous Computing,* pp. 81-84, 2017.

[74] Raspberry Pi Foundation, "Raspberry Pi," [Online]. Available: https://www.raspberrypi.org/. [Accessed October 2020].

[75] S. Li, L. Da Xu and S. Zhao, "The Internet of Things: A Survey," *Information Systems Frontiers,* vol. 17, no. 2, pp. 243-259, 2015.

[76] R. Khan, S. U. Khan, R. Zaheer and S. Khan, "Future Internet: The Internet of Things architecture, possible applications and key challenges," *Frontiers of Information Technology,* pp. 257-260, 2012.

[77] A. Khaled, W. Lindquist and S. Helal, "DIY Health IoT Apps," *ACM Conference on Embedded Network Systems,* pp. 406-407, November 2018.

[78] P. Bratley and J. Millo, "Computer Recreations," *Software – Practice and Experience,* vol. 2, pp. 397-400, 1972.

[79] S. Helal, A. E. Khaled and V. Gutta, "Demo: Atlas Thing Architecture: Enabling Mobile Apps as Things in the IoT.," *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking (MobiCom '17),* pp. 480-482, 2017.

[80] Google, "Android Toast Overview," [Online]. Available: https://developer.android.com/guide/topics/ui/notifiers/toasts. [Accessed October 2020].

[81] Android Developer Documentation, "WebView," [Online]. Available: https://developer.android.com/reference/android/webkit/WebView. [Accessed October 2020].

[82] Android Developer Documentation, "Intent," [Online]. Available: https://developer.android.com/reference/android/content/Intent. [Accessed October 2020].

[83] Android Developer Documentation, "Input Controls," [Online]. Available: https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/. [Accessed October 2020].

[84] "Semantic UI," [Online]. Available: https://semantic-ui.com/. [Accessed October 2020].

[85] Ajax.org, "Ace Editor," [Online]. Available: https://ace.c9.io/. [Accessed October 2020].

[86] Android Developer Documentation, "LinearLayout," [Online]. Available: https://developer.android.com/reference/android/widget/LinearLayout. [Accessed October 2020].

[87] Google, "Download Android Studio," [Online]. Available: https://developer.android.com/studio. [Accessed October 2020].

[88] G. A. Miller, "WordNet: a lexical database for English," *Communications of the ACM,* vol. 38, no. 11, pp. 39-41, 1995.

[89] Princeton University, "WordNet," [Online]. Available: https://wordnet.princeton.edu/. [Accessed October 2020].

[90] J. Ramos, "Using TF-IDF to Determine Word Relevance in Document Queries," in *The First Instructional Conference on Machine Learning*, 2003.

[91] Google, "Android Services Overview," [Online]. Available: https://developer.android.com/guide/components/services. [Accessed October 2020].

[92] Google, "Work with Observable Data Objects," [Online]. Available: https://developer.android.com/topic/libraries/data-binding/observability. [Accessed October 2020].

[93] Android Developer Documentation, "ViewGroup," [Online]. Available: https://developer.android.com/reference/android/view/ViewGroup. [Accessed October 2020].

[94] JetBrains, "IntelliJ IDEA Intention Actions," [Online]. Available: https://www.jetbrains.com/help/idea/intention-actions.html. [Accessed October 2020].

[95] Intel, "Intel Edison Product Brief," [Online]. Available: https://www.intel.com/content/dam/ support/us/en/documents/ edison/sb/edison_pb_331179002.pdf. [Accessed October 2020].

[96] CppMicroServices, "C++ Micro Services," [Online]. Available: http://cppmicroservices.org/. [Accessed October 2020].

[97] Google, "Introduction to Android Activities," [Online]. Available: https://developer.android.com/guide/components/activities/intro-activities. [Accessed November 2020].

[98] C. Lallemand and G. Gronier, "Enhancing User eXperience during waiting time in HCI: Contributions of cognitive psychology," in *Proceedings of the Designing Interactive Systems Conference, DIS'12*, 2012.

[99] F. Nah, "A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait?," *Behaviour & Information Technology - Behaviour & IT,* vol. 23, no. 285, 2003.

[100] Google, "Android Source Tree - config.xml," 2019. [Online]. Available: android.googlesource.com/platform/frameworks/base/+/

refs/heads/master/core/res/ res/values/config.xml. [Accessed December 2019].

[101] Android Developer Documentation, "System," [Online]. Available: https://developer.android.com/reference/java/lang/System. [Accessed November 2020].

[102] cppreference.com, "std::chrono::system_clock," January 2020. [Online]. Available: https://en.cppreference.com/w/cpp/chrono/system_clock. [Accessed November 2020].

[103] T. Tullis and W. Albert, "Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics," 2008.

[104] iHealth Labs, "iHealth Air Pulse Oximeter," [Online]. Available: https://ihealthlabs.com/product/ihealth-air-po3/. [Accessed November 2020].

[105] Withings, "Under-Mattress Sleep Tracker," [Online]. Available: https://www.withings.com/uk/en/sleep-analyzer. [Accessed November 2020].

[106] Google, "Profile Battery Usage with Batterystats and Battery Historian," [Online]. Available: https://developer.android.com/topic/ performance/power/setup- battery- historian. [Accessed November 2020].

[107] Wellue, "O2Ring Continuous Ring Oximeter," [Online]. Available: https://getwellue.com/pages/o2ring-oxygen-monitor. [Accessed November 2020].

[108] Nordic Semiconductor, "nRF52840 Bluetooth 5.2 SoC," [Online]. Available: https://www.nordicsemi.com/Products/ Low-power-short-range-wireless/nRF52840. [Accessed November 2020].

APPENDIX A: IOT-DDL SCHEMA DEFINITION

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:annotation><xs:documentation>
        This document describes the schema for the IoT Device Description
        Language (IoT-DDL). This includes functional attributes (services,
        network properties, etc.), as well as a variety of metadata fields.
    </xs:documentation></xs:annotation>

    <xs:simpleType name="identifier">
        <xs:annotation><xs:documentation>
            An identifier is used to specify the machine-usable name of the
            thing and space, as well as services, unbounded services, and
            relationships. An identifier is almost always paired with a human-
            readable identifier, called the "Name".
        </xs:documentation></xs:annotation>
        <xs:restriction base="xs:string">
            <xs:pattern value="[a-zA-Z0-9_]+" />
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="range">
        <xs:sequence>
            <xs:element name="Lower_Bound" type="xs:integer" />
            <xs:element name="Upper_Bound" type="xs:integer" />
        </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="ipAddress">
        <xs:restriction base="xs:string">
            <xs:pattern value="((25[0-5]|2[0-4][0-9]|1?[0-9][0-9]?)(\.|$)){4}" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="address">
        <xs:union memberTypes="xs:anyURI ipAddress" />
    </xs:simpleType>

    <xs:simpleType name="port">
        <xs:restriction base="xs:positiveInteger">
            <xs:maxInclusive value="65535" />
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="endpoint">
        <xs:sequence>
            <xs:element name="Address" type="address" />
            <xs:element name="Port" type="port" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="authEndpoint"><xs:complexContent>
        <xs:extension base="endpoint"><xs:sequence>
            <xs:element name="Username" type="xs:string" />
            <xs:element name="Password" type="xs:string" />
        </xs:sequence></xs:extension>
    </xs:complexContent></xs:complexType>

    <xs:simpleType name="path">
        <xs:restriction base="xs:string">
            <xs:pattern value="(\/[a-zA-Z0-9_])+" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="keywords">
        <xs:annotation><xs:documentation>
            Descriptive keywords are a core part of the IoT-DDL functionality.
            These keywords are used to match service and relationship
```

```
            functionality, as well as to enable features such as MAAT. Because
            a full keyword match looks for similar words and uses IDF, this
            list should usually be short (less than 4).
        </xs:documentation></xs:annotation>
        <xs:restriction base="xs:string">
            <xs:pattern value="([a-z]+(\.|$))+" />
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="interval">
        <xs:sequence>
            <xs:element name="Start_Time" type="xs:time" />
            <xs:element name="End_Time" type="xs:time" />
        </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="dataType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="integer" />
            <xs:enumeration value="float" />
            <xs:enumeration value="string" />
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="parameter">
        <xs:sequence>
            <xs:element name="Description" type="xs:string" />
            <xs:element name="Type" type="dataType" />
            <xs:element name="Range" type="range" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="hexadecimal">
        <xs:restriction base="xs:string">
            <xs:pattern value="0x[0-9A-F]+" />
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="metadata">
        <xs:annotation><xs:documentation>
            Most types contain some form of metadata - therefore, many fields
            extend from this type. That is, entities, services, relationships,
            etc. will all require a name, ID, and optional description. Note
            that any references to other elements (such as in relationships)
            will always refer to the ID, not the Name.
        </xs:documentation></xs:annotation>
        <xs:sequence>
            <xs:element name="ID" type="identifier" />
            <xs:element name="Name" type="xs:string" />
            <xs:element name="Description" type="xs:string" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="partMetadata"><xs:complexContent>
        <xs:annotation><xs:documentation>
            Physical parts (the thing and its entities) use this metadata.
        </xs:documentation></xs:annotation>
        <xs:extension base="metadata"><xs:sequence>
            <xs:element name="Owner" type="xs:string" />
            <xs:element name="Vendor" type="xs:string" />
        </xs:sequence></xs:extension>
    </xs:complexContent></xs:complexType>

    <xs:simpleType name="thingType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Platform Thing" />
            <xs:enumeration value="Sensor Mote Thing" />
            <xs:enumeration value="Bit Thing" />
            <xs:enumeration value="Soft Thing" />
            <xs:enumeration value="Edge Thing" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="thingOS">
        <xs:annotation><xs:documentation>
```

```
            This enumeration represents the current platform support (full or
            partial) the architecture offers.
        </xs:documentation></xs:annotation>
        <xs:restriction base="xs:string">
            <xs:enumeration value="Raspberry Linux" />
            <xs:enumeration value="Debian Linux" />
            <xs:enumeration value="mbedOS" />
            <xs:enumeration value="Android" />
            <xs:enumeration value="Arduino" />
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="thingMetadata"><xs:complexContent>
        <xs:extension base="partMetadata"><xs:sequence>
            <xs:element name="Model" type="xs:string" />
            <xs:element name="Release_Date" type="xs:date" />
            <xs:element name="Type" type="thingType" />
            <xs:element name="Operating_System" type="thingOS" />
        </xs:sequence></xs:extension>
    </xs:complexContent></xs:complexType>

    <xs:complexType name="spaceConstraints">
        <xs:annotation><xs:documentation>
            Space constraints currently are mostly metadata/descriptive.
        </xs:documentation></xs:annotation>
        <xs:sequence>
            <xs:element name="Temperature" type="range" />
            <xs:element name="Humidity" type="range" />
            <xs:element name="Voltage" type="range" />
            <xs:element name="Interference_Radius" type="xs:positiveInteger" />
            <xs:element name="Temperature_Radius" type="xs:positiveInteger" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="spaceMetadata"><xs:complexContent>
        <xs:extension base="metadata"><xs:sequence>
            <xs:element name="Constraints" type="spaceConstraints" />
        </xs:sequence></xs:extension>
    </xs:complexContent></xs:complexType>

    <xs:complexType name="atlasMetadata">
        <xs:sequence>
            <xs:element name="Thing_Metadata" type="thingMetadata" />
            <xs:element name="Space_Metadata" type="spaceMetadata" />
        </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="networkModule">
        <xs:restriction base="xs:string">
            <xs:enumeration value="WiFi" />
            <xs:enumeration value="Ethernet" />
            <xs:enumeration value="Bluetooth" />
            <xs:enumeration value="Bluetooth Low Energy" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="networkType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Built-In" />
            <xs:enumeration value="Attached" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="networkProtocol">
        <xs:restriction base="xs:string">
            <xs:enumeration value="IP" />
            <xs:enumeration value="MQTT" />
            <xs:enumeration value="REST" />
            <xs:enumeration value="CoAP" />
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="networkMetadata">
        <xs:sequence>
            <xs:element name="Module" type="networkModule" />
```

```
        <xs:element name="Type" type="networkType" />
        <xs:element name="Protocol" type="networkProtocol" />
        <xs:element name="WiFi_SSID" type="xs:string" minOccurs="0" />
        <xs:element name="WiFi_Password" type="xs:string" minOccurs="0" />
        <xs:element name="Multicast_Group" type="endpoint" />
    </xs:sequence>
</xs:complexType>

<xs:complexType name="atlasThing">
    <xs:sequence>
        <xs:element name="Descriptive_Metadata" type="atlasMetadata" />
        <xs:element name="Administrative_Metadata" type="networkMetadata" />
    </xs:sequence>
</xs:complexType>

<xs:complexType name="protocolTopics">
    <xs:sequence>
        <xs:element name="Root" type="path" />
        <xs:element name="MQTT_Client" type="path" />
        <xs:element name="Private_Broker" type="path" />
        <xs:element name="Tweet_ThingIdentity" type="path" />
        <xs:element name="Tweet_EntityIdentity" type="path" />
        <xs:element name="API" type="path" />
        <xs:element name="Interaction" type="path" />
    </xs:sequence>
</xs:complexType>

<xs:complexType name="protocolTranslator">
    <xs:sequence>
        <xs:element name="Broker" type="authEndpoint" />
        <xs:element name="Topic" type="protocolTopics" />
    </xs:sequence>
</xs:complexType>

<xs:complexType name="attachments">
    <xs:annotation><xs:documentation>
        The attachment section is optional; it contains endpoints to
        external thing services that are on offer in the smart space. Of
        interest is the App Generator, which is used for the DIY Health Apps
        demo and examples.
    </xs:documentation></xs:annotation>
    <xs:sequence>
        <xs:element name="App_Generator" type="endpoint" minOccurs="0" />
        <xs:element name="Log_Server" type="authEndpoint" minOccurs="0" />
        <xs:element name="Protocol_Translator" type="protocolTranslator" minOccurs="0" />
        <xs:element name="OMA_DM" type="authEndpoint" minOccurs="0" />
    </xs:sequence>
</xs:complexType>

<xs:simpleType name="entityCategory">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Software" />
        <xs:enumeration value="Hardware" />
        <xs:enumeration value="Hybrid" />
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="entityType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="Built-In" />
        <xs:enumeration value="Connected" />
        <xs:enumeration value="Remote" />
    </xs:restriction>
</xs:simpleType>

<xs:complexType name="entityMetadata"><xs:complexContent>
    <xs:extension base="partMetadata"><xs:sequence>
        <xs:element name="Category" type="entityCategory" />
        <xs:element name="Type" type="entityType" />
    </xs:sequence></xs:extension>
</xs:complexContent></xs:complexType>

<xs:simpleType name="proxyProtocol">
    <xs:restriction base="xs:string">
        <xs:enumeration value="BLE" />
```

```
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="proxyAddress">
        <xs:restriction base="xs:string">
            <xs:pattern value="([0-9A-F]{2}(\:|$)){6}" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="proxyAccess">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Async" />
            <xs:enumeration value="Write" />
            <xs:enumeration value="Notify" />
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="proxyAttribute">
        <xs:sequence>
            <xs:element name="Name" type="xs:string" />
            <xs:element name="Handle" type="hexadecimal" />
            <xs:element name="Access" type="proxyAccess" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="proxyInterface">
        <xs:annotation><xs:documentation>
            A proxy interface (currently only BLE) represents an external
            device that is incapable of running the thing architcture itself.
            This proxy-ing thing will instead expose the listed interfaces as
            its own services for use internally / by other things.
        </xs:documentation></xs:annotation>
        <xs:sequence>
            <xs:element name="Protocol" type="proxyProtocol" />
            <xs:element name="Address" type="proxyAddress" />
            <xs:element name="Attribute" type="proxyAttribute" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="proxyInterfaces">
        <xs:sequence>
            <xs:element name="Proxy_Interface" type="proxyInterface" minOccurs="0"
maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="serviceType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Action" />
            <xs:enumeration value="Report" />
            <xs:enumeration value="Condition" />
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="contextConstraints">
        <xs:sequence>
            <xs:element name="Interval" type="xs:integer" />
            <xs:element name="Concurrent" type="xs:boolean" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="timeConstraints">
        <xs:sequence>
            <xs:element name="Working" type="interval" />
            <xs:element name="Callable" type="interval" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="serviceInputs">
        <xs:sequence>
            <xs:element name="Input" type="parameter" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="service"><xs:complexContent>
```

```
        <xs:annotation><xs:documentation>
            A service's entire description is metadata, however, it is also
            used to construct the service runtimes for the thing. The fields
            used for this are "Input", "Output", and "Formula".
        </xs:documentation></xs:annotation>
        <xs:extension base="metadata"><xs:sequence>
            <xs:element name="Category" type="xs:string" />
            <xs:element name="Type" type="serviceType" />
            <xs:element name="Keywords" type="keywords" />
            <xs:element name="Contextual_Constraints" type="contextConstraints" />
            <xs:element name="Temporal_Constraints" type="timeConstraints" />
            <xs:element name="Inputs" type="serviceInputs" />
            <xs:element name="Output" type="parameter" minOccurs="0" />
            <xs:element name="Formula" type="xs:string" />
        </xs:sequence></xs:extension>
    </xs:complexContent></xs:complexType>

    <xs:complexType name="entityServices">
        <xs:sequence>
            <xs:element name="Service" type="service" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="unbounded"><xs:complexContent>
        <xs:annotation><xs:documentation>
            An unbounded service represents an external thing service (different
            from a proxy interface, however, as this is a service running on a
            device with the thing architecture). If any metadata of shared
            services match, it can "bind", allowing it to be used by other
            things and smart space users in relationships, etc.
        </xs:documentation></xs:annotation>
        <xs:extension base="metadata"><xs:sequence>
            <xs:element name="Type" type="serviceType" />
            <xs:element name="Vendors" type="keywords" minOccurs="0" />
            <xs:element name="Keywords" type="keywords" minOccurs="0" />
            <xs:element name="Match_Value" type="xs:positiveInteger" />
        </xs:sequence></xs:extension>
    </xs:complexContent></xs:complexType>

    <xs:complexType name="entityUnboundeds">
        <xs:sequence>
            <xs:element name="Unbounded_Service" type="unbounded" minOccurs="0"
maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="relationshipType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Control" />
            <xs:enumeration value="Drive" />
            <xs:enumeration value="Support" />
            <xs:enumeration value="Extend" />
            <xs:enumeration value="Contest" />
            <xs:enumeration value="Interfere" />
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="relationship"><xs:complexContent>
        <xs:annotation><xs:documentation>
            A relationship is a metadata construct that defines links between
            (unbounded) services to be used by external applications.
        </xs:documentation></xs:annotation>
        <xs:extension base="metadata"><xs:sequence>
            <xs:element name="Establisher" type="xs:string" />
            <xs:element name="Type" type="relationshipType" />
            <xs:element name="Service_LHS" type="identifier" />
            <xs:element name="Service_RHS" type="identifier" />
        </xs:sequence></xs:extension>
    </xs:complexContent></xs:complexType>

    <xs:complexType name="entityRelationships">
        <xs:sequence>
            <xs:element name="Relationship" type="relationship" minOccurs="0"
maxOccurs="unbounded" />
        </xs:sequence>
```

```
    </xs:complexType>

    <xs:complexType name="entity">
        <xs:annotation><xs:documentation>
            An entity represents an entire "bundle" of service behaviors for a
            single feature of the thing. All services within an entity share a
            "context" of data and computation. Before services are created,
            shared state can be set up in the "Entrypoint", which is a code
            block in the style of a service's formula.
        </xs:documentation></xs:annotation>
        <xs:sequence>
            <xs:element name="Descriptive_Metadata" type="entityMetadata" />
            <xs:element name="Entrypoint" type="xs:string" />
            <xs:element name="Proxy_Interfaces" type="proxyInterfaces" minOccurs="0" />
            <xs:element name="Services" type="entityServices" minOccurs="0" />
            <xs:element name="Unbounded_Services" type="entityUnboundeds" minOccurs="0" />
            <xs:element name="Relationships" type="entityRelationships" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="entities">
        <xs:sequence>
            <xs:element name="Entity" type="entity" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="ddl">
        <xs:sequence>
            <xs:element name="Atlas_Thing" type="atlasThing" />
            <xs:element name="Thing_Attachments" type="attachments" minOccurs="0" />
            <xs:element name="Atlas_Entities" type="entities" />
        </xs:sequence>
    </xs:complexType>

    <xs:element name="IoT_DDL" type="ddl" />

</xs:schema>
```

APPENDIX B: MAAT USER STUDY CONSENT FORM

LANCASTER UNIVERSITY

Research Participant Consent Form

PROJECT TITLES: MAAT Plugin USABILITY STUDY

INVESTIGATORS: Wyatt Lindquist

PARTICIPANT NAME: _____

TITLE: _____

I agree to participate in the project named above, the particulars of which have been explained to me. I have read the Research Project Description a written copy of which has been given to me to keep.

I understand that any information I provide is confidential, and that, subject to the limitations of the law, no information that could lead to the identification of any individual will be directly disclosed in any reports on the project, or to any other party.

I agree to being: (tick as appropriate):

- ☐ observed;
- ☐ interviewed;
- ☐ part of a focus group (questionnaire)

I agree to the following data being collected:

- ☐ Time taken for me to complete specified tasks
- ☐ The amount of task completed (i.e. percent of task compled)
- ☐ Number of errors made per task;
- ☐ Number of times I request for help to complete a task
- ☐ Time needed for me to work around a task

I also agree to the data above being used for later analysis by the researchers above only. To preserve anonymity, I understand that all written work referring to this data will use pseudonyms for me unless written permission is later obtained. I also understand that direct access to the identity of participants is restricted to named researchers above only.

School of Computing and Communications

School of Computing and Communications | InfoLab21 | Lancaster University | LA1 4WA
Tel: 01524 510302 | Email: scc@lancaster.ac.uk
www.scc.lancs.ac.uk

I acknowledge that:

a) I have been informed that I am free to withdraw from the project at any time without explanation or prejudice and to withdraw data previously supplied.
b) Participation in this project is voluntary.

Signature: _____ Date: _____
(Participant)

School of Computing and Communications

School of Computing and Communications | InfoLab21 | Lancaster University | LA1 4WA
Tel: 01524 510302 | Email: scc@lancaster.ac.uk
www.scc.lancs.ac.uk