King's Research Portal

DOI:
10.1109/eScience55777.2022.00019

Document Version
Peer reviewed version

Link to publication record in King's Research Portal

Citation for published version (APA):
Chapman, M., G-Medhin, A., Sassoon, I., Kokciyan, N., Sklar, E., & Curcin, V. (2022). Using Microservices to Design Patient-facing Research Software. In *IEEE 18th International Conference on eScience* https://doi.org/10.1109/eScience55777.2022.00019

Citing this paper
Please note that where the full-text provided on King's Research Portal is the Author Accepted Manuscript or Post-Print version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version for pagination, volume/issue, and date of publication details. And where the final published version is provided on the Research Portal, if citing you are again advised to check the publisher's website for any subsequent corrections.

General rights
Copyright and moral rights for the publications made accessible in the Research Portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognize and abide by the legal requirements associated with these rights.

•Users may download and print one copy of any publication from the Research Portal for the purpose of private study or research.
•You may not further distribute the material or use it for any profit-making activity or commercial gain
•You may freely distribute the URL identifying the publication in the Research Portal

Take down policy
If you believe that this document breaches copyright please contact librarypure@kcl.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 13. Jan. 2025

Using Microservices to Design Patient-facing Research Software

Martin Chapman
Department of Population Health Sciences
King's College London
London, UK
martin.chapman@kcl.ac.uk

Abigail G-Medhin
Department of Population Health Sciences
King's College London
London, UK
abigail.g-medhin@kcl.ac.uk

Isabel Sassoon
Department of Computer Science
Brunel University London
Uxbridge, UK
isabel.sassoon@brunel.ac.uk

Nadin Kökciyan
School of Informatics
University of Edinburgh
Edinburgh, UK
nadin.kokciyan@ed.ac.uk

Elizabeth I. Sklar
Lincoln Institute for Agri-food Technology
University of Lincoln
Lincoln, UK
esklar@lincoln.ac.uk

Vasa Curcin
Department of Population Health Sciences
King's College London
London, UK
vasa.curcin@kcl.ac.uk

***Abstract*—With a significant amount of software now being developed for use in patient-facing studies, there is a pressing need to consider how to design this software effectively in order to support the needs of both researchers and patients. We posit that a microservice architecture—which offers a large amount of flexibility for development and deployment, while at the same time ensuring certain quality attributes, such as scalability, are present—provides an effective mechanism for designing such software. To explore this proposition, in this work we show how the paradigm has been applied to the design of CONSULT, a decision support system that provides autonomous support to stroke patients and is characterised by its use of a data-backed AI reasoner. We discuss the impact that the use of this software architecture has had on the teams developing CONSULT and measure the performance of the system produced. We show that the use of microservices can deliver software that is able to facilitate both research and effective patient interactions. However, we also conclude that the impact of the approach only goes so far, with additional techniques needed to address its limitations.**

***Index Terms*—Microservice architectures, Decision support systems, Artificial intelligence**

I. INTRODUCTION

Traditionally, research software is designed to provide functionality to researchers themselves, who may, for example, use a piece of software to simulate a given phenomenon and directly measure the outputs of the software in order to extract some scientific understanding. However, increasingly, research software is also being presented to end-users where, instead, scientific understanding is extracted from the way in which software is able to support these users. This is typified by research in the field of *health informatics* (HI), where these users are patients, and the development of software to support translational and clinical research is commonplace. For example, a component of a piece of research software developed in HI may be a healthcare dialogue system that is designed to use *artificial intelligence* (AI) to assist in decision-making and understanding is extracted from how that system is able to interact with a patient. Research such as this is ever-increasing, given the pressing need to personalise and automate healthcare provision to large cohorts of patients with complex needs.

Developing a piece of software for patient-based research brings a number of technical challenges, borne from the need to support the requirements of both researchers and patients. These include:

- **Software development**—Typically, the programming languages used by researchers to develop software are those that prioritise: (a) a low barrier to development entry (to enable domain researchers, who are not necessarily trained software engineers, to realise their contributions) and (b) technical package support (to enable the sharing and reuse of contributions), e.g. Python. However, those languages best suited to the development of patient-facing software are typically those that prioritise: (a) scalability (e.g. V8-compiled Javascript) in order to ensure access is available to potentially large groups of users and (b) user interface design in order to ensure that patients are exposed to accessible clients.
- **Modularity**—In order to experiment with different approaches, researchers often want to change various components within a system and observe the effects. When applying technologies like AI, this may be particularly prevalent when there are competing approaches to the same tasks (e.g. different classification models trained by different machine learning algorithms). Typically, one would not expect such changes to be made in a patient-facing system, where the overall stability and availability of the platform is essential.
- **Reproducibility**—Researchers are keen for their results to be reproducible, and as such the software from which

This work was supported by the UK Engineering & Physical Sciences Research Council (EPSRC), grant #EP/P010105/1.

those results have been derived should be available and easy for others to run. However, when a piece of software contains logic relating to the processing of patient data, it may not, for security reasons, be possible for that system to be shared.
- **Processing time**—Researchers often want to evaluate components that are associated with longer execution times (e.g. a classification model may take many cycles to produce predictions when examining a large amount of data). In contrast, the components of a patient-facing system need to execute in as near real-time as possible, in order to ensure that potentially critical responses are returned to users in sufficient time.
- **Data quality**—The data that researchers wish to use as input to their software is likely to be high-volume, heterogeneous, stream intermittently and originate from a variety of different sources. Within a patient-facing system, data sources need to be as accurate and stable as possible, in order to ensure any health interventions are based on correct information.

To address these challenges, we propose that patient-facing research software be developed according to the *microservice* architecture paradigm. To support this proposal, in this paper we detail how CONSULT—an AI-based decision support system (DSS) created to facilitate research into the *self-management* of chronic health conditions such as stroke—has been designed according to this paradigm, and discuss the benefits this has brought. As a component of this discussion, we quantify the impact of the microservice approach by evaluating the performance of CONSULT using a number of different metrics. We also introduce REFLECT, a project that seeks to explore the impact of, and attitudes towards, the use of wearable data within decision support, and was made possible by the design of CONSULT.

II. BACKGROUND AND RELATED WORK

A. Microservices

The primary principle of the microservice architecture paradigm is that traditional monolithic architectures should instead be separated into individual communicating services, each of which provides a single piece of overall system functionality [1]. Although it has its roots in more traditional service-oriented architectures, the microservice approach is derived from the use of services in practice, which has led to a more *opinionated* paradigm that emphasises the importance of clear service boundaries and limited-service scope. Microservices also differ from traditional service-oriented architectures in the type of communication used between services. Typically HTTP calls to well-defined REST interfaces provide communication between services [2], but other mechanisms can also be leveraged, such as publish/subscribe (pub/sub) models, and queue-based implementations.

In industry, the use of microservices is now commonplace, with one of the most prolific and early adopters of the architecture being the Netflix streaming service [3]. In the research domain, there are also several examples of the use of microservices, typically to support researchers directly in various tasks. *NDStore* is a storage and analysis platform for brain imaging data, which is realised using microservice technologies [4]. Different services exist to support data ingest, and analysis workflows. Similarly, *OceanTEA* is a software platform that supports collaborative, multi-platform analysis of oceanographic data, and is also realised as a microservice architecture [5]. Based on the popularity of using the approach to design data science platforms, the *GeRDI* project tries to abstract research software architectures like the ones described into a high-level architecture that consists of a set of common service types [6]. This architecture can then be used to guide the creation of services when future platforms are developed. The services represented include those supporting the collection of metadata, data pre-processing and the data analysis itself. However, Schröer warns that a 'one-size-fits-all' mapping, such as this—between the activities found within a data science workflow (i.e. data retrieval through to analysis and publication) and the services that support that workflow—might not be feasible, given the fact that in some domains certain task are more tightly related than in others [7]. For example, in one domain data preparation and analytics might be very distinct research processes, and thus be better supported by a piece of software comprising a service responsible for each, but in another these tasks might essentially be joined, and thus be better supported by a single service. Overall, this shows us that determining the remit of a service is a difficult task, and should be driven by the domain a system is being designed for.

In the health informatics research domain specifically, microservices have been used to support researchers in a number of areas, including the collection of patient data. Garcia-Moreno et al. use a microservice architecture to collect data from a single wearable device and use it to train a frailty classification model [8]. The performance of the classifier is then assessed offline. In the same way, Roca et al. have developed a chatbot, supported by a microservice architecture, which gathers data from patients with chronic conditions (such as images from psoriasis patients) [9]. Although the system has clinical applications, the focus on the gathering of patient data suggests that, at present, the platform is likely to be used to support activities such as epidemiological research. The work presented here demonstrates the use of a microservice architecture in the HI domain to facilitate a clinical, patient-facing application.

B. *The* CONSULT *decision support system*

With the increasing strain on healthcare resources, the presence of digital support tools within routine clinical care settings is on the rise. A common support tool is a (clinical) *decision support system* (DSS), a piece of software that stores clinical knowledge and patient-specific data, reasons with that knowledge and data using patient preferences, and provides output to a clinician, assisting their practice. Such tools can also be accessed directly by patients, to facilitate the self-

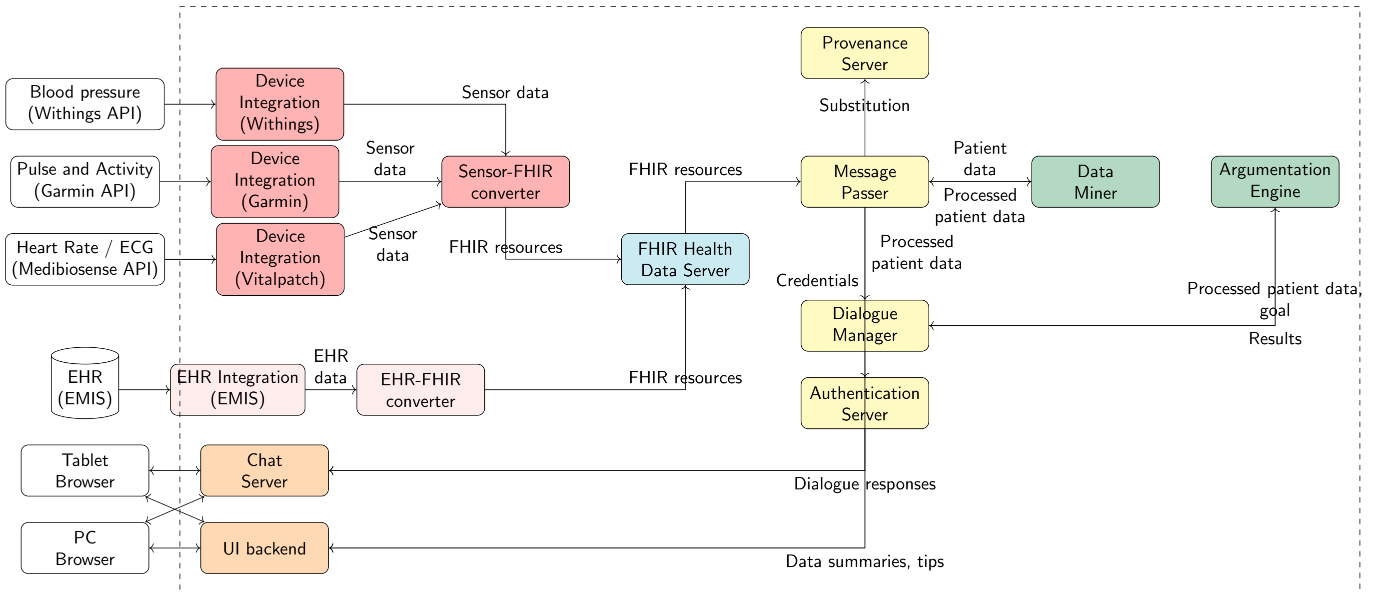


Fig. 1: CONSULT's microservice architecture

management of health and wellness, further reducing the burden on professional healthcare services.

Due to their potential impact, several research studies have explored the impact of DSSs on healthcare provision. The TRANSFoRm project looked at implementing a DSS that integrates with local EHR systems in order to provide clinicians with support directly at the point of care [10]. Specifically, to assist with a clinician's workflow, EHR data is mined in order to display potential diagnoses when a patient presents with a certain condition. To support more long-term care, TRANSFoRm also explored the potential for using a DSS for clinical trial recruitment at the point of care. In much the same way, the DSS developed as a part of the HARVEST project aims to support a clinician in understanding a patient's medical record by summarising its content using Natural Language Processing (NLP) and visualisation techniques [11]. For example, salient conditions are presented to a clinician by extracting key terms from a patient's medical record and comparing how frequently they appear within a given time period (selected by the clinician) to how frequently they appear in the records of other patients at the same practice.

Despite the research that has been undertaken into DSSs, several challenges remain, including how to support patients with complex needs, such as those with the multiple morbidities often associated with chronic conditions like stroke. With this challenge in mind, the goal of the CONSULT DSS is to combine several patient-facing DSS components (e.g. a traditional user interface (UI) and a chatbot) with a *computational argumentation-based* AI engine [12]. *Computational Argumentation* is a logic-based methodology in which claims are presented as *evidence*, either supporting or attacking specific *conclusions*, and it thus maps well to the medical decision-making domain [13]. To ensure that the argumentation engine has a holistic, up-to-date view of a patient, it has access to data gathered from wearable wellness devices (which patients can also view directly) and their *electronic health record (EHR)*. These inputs are used to personalise and autonomously reason with stored clinical knowledge in order to provide transparent healthcare recommendations [14], [15].

A prototype of the CONSULT system has been co-designed with patients, implemented and demonstrated [16], [17]. This system integrates with several commercial off-the-shelf wellness sensors, including a blood pressure monitor (Withings[1]), heart rate tracker (Garmin[2]) and the *Vitalpatch* ECG monitor (Medibiosense[3]), a low-cost bespoke sensor. Wellness sensor integration was achieved by connecting with the APIs exposed by each vendor. A proof-of-concept (PoC) integration was also developed for EMIS[4], a UK EHR company. CONSULT has undergone a successful pilot study with a number of users [18], and is shortly due for patient research trials.

III. SYSTEM DESIGN

As a result of applying the principles of microservice design to CONSULT, its overall system architecture is shown in Fig. 1. Here, the services that comprise the system are shown, along with system inputs. While each service has a specific function, which is described by the label provided, services can also be conceptually grouped according to their collective functionality: services designed to collect and format sensor data (red); to collect EHR data (pink); to store sensor data (specifically in the FHIR health data standard [19]) (blue); to co-ordinate the flow of data through the system (yellow); to interface with the user (orange); and to perform background processing tasks (green), including computational argumentation-based reasoning. Each

[1] https://withings.com
[2] https://garmin.com
[3] https://medibiosense.com/vitalpatch
[4] https://www.emishealth.com

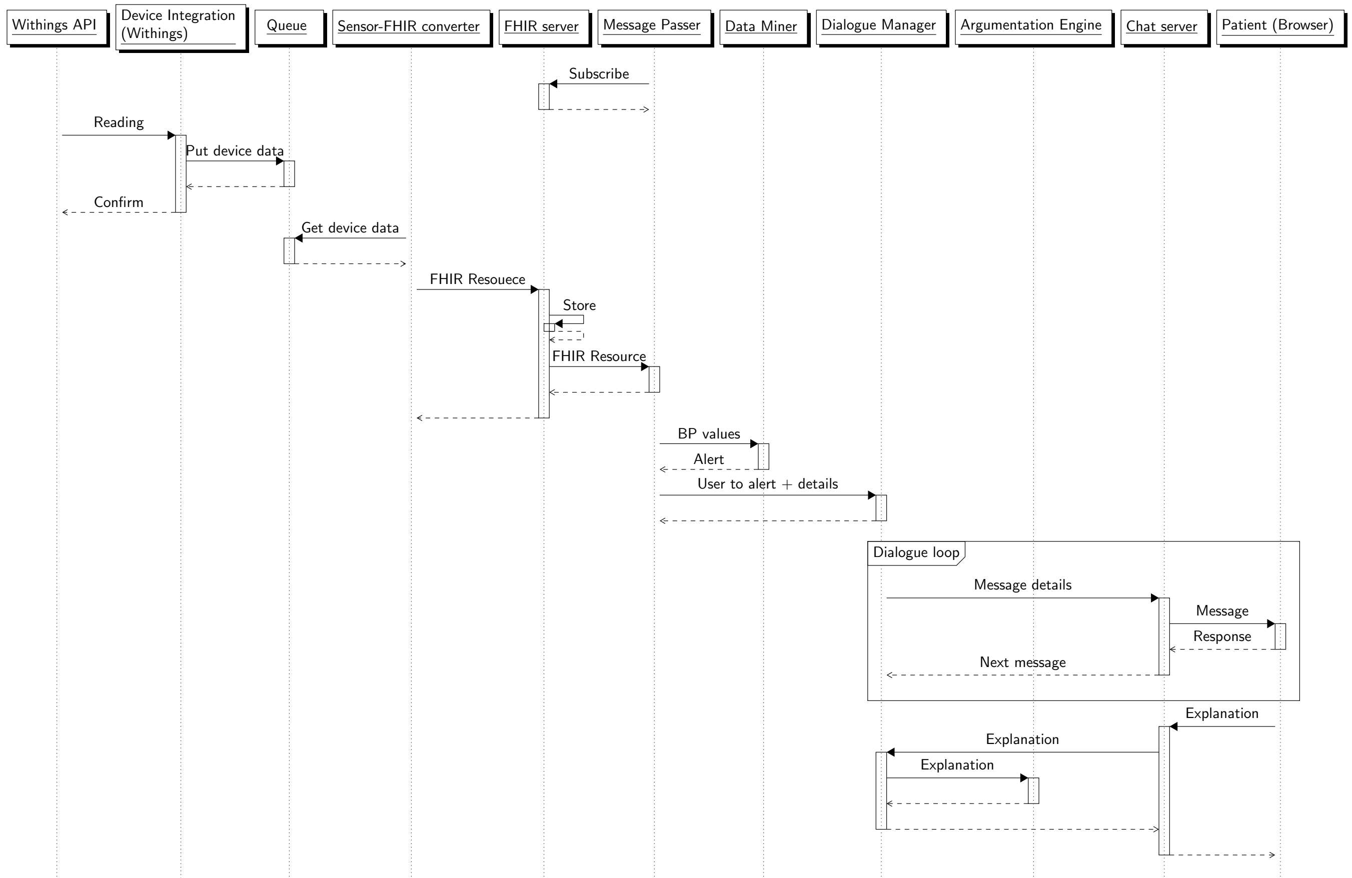


Fig. 2: CONSULT services combining to provide alert functionality

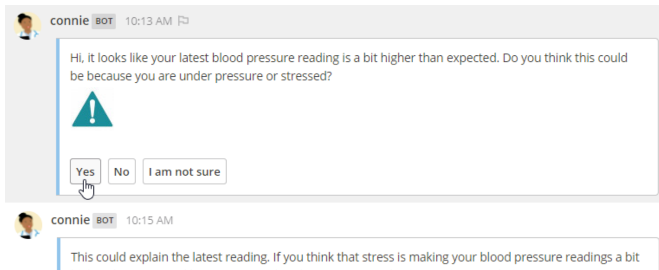
service can operate as a self-contained entity, encapsulating its processing logic behind a well-defined REST or queue-based interface, allowing the functionality it offers to be invoked by other services.

During the software development cycle of the project, the functionality of each service was determined iteratively, starting with a coarser scope and then refining the remit of each service to a suitable granularity. This process was guided by a domain-drive design approach, specifically the notion of *bounded context*, the idea that the encapsulation found in a given domain (e.g. the unique responsibilities held by different departments in a commercial organisation) could and should be reflected in the services that support the work done in that domain [20]. This is evident, for example, in the choice to clearly separate the reasoning components from the data processing components, which are development areas in which different research function-focussed sub-groups within the greater CONSULT project worked.

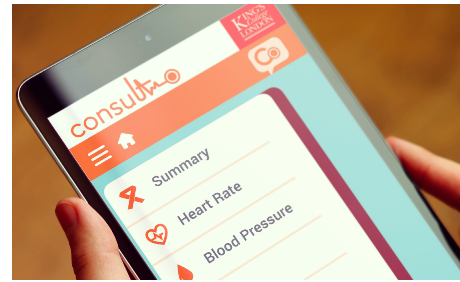
In practice, each service is a containerised (Dockerised) application, typically paired with both an (API) gateway (e.g. Nginx) and a database (e.g. MariaDB). The data stored by each service in its database depends on its functionality. For some services this may be logging information, but for others, such as the FHIR health data service, this is specific (encrypted) data such as EHR data combined with wearables data. Containers are either orchestrated directly by Docker compose[5] or managed by Kubernetes[6]. Based upon their service category (Fig. 1), these containers are split over different virtual machines, each accessible via a public subnet or only available within a private subnet, depending on the functionality being offered. This allows us to, for example, place the health data service on its own machine—to which additional security measures are applied—within a private subnet, while still having our user-facing services accessible. Virtual machines (nodes) are provided by Amazon Web Services (AWS), and range in capacity from 1 (v)CPU, 500MB RAM to 4 (v)CPUs, 16GB RAM, depending on the services that are present on the node.

Together, the services in CONSULT communicate to provide individual pieces of functionality within the system. Communication is either via HTTP, a queue-based intermediary or via a pub/sub mechanism. An example of this communication is shown in the sequence diagram in Fig. 2. Here, the system is responding to an exacerbation in a patient's blood pressure, as read in real-time from the Withings wearable blood pressure sensor, which a patient has previously registered with the system. This information is passed from the device

[5]https://github.com/docker/compose
[6]https://kubernetes.io



(a) The chatbot interacting with a user



(b) Design of the CONSULT dashboard

Fig. 3: CONSULT's chatbot and dashboard interfaces

manufacturer's server (e.g. Withings) to the CONSULT system, and then passed through various services to standardise, store and analyse the reading, before the alert is generated and sent to the user via a chatbot (Fig. 3). Evidence of the alert is also present in a UI dashboard. The user can then respond to this alert by asking for more information, which is then provided by the computational argumentation engine and communicated to the user, structured as a *computational argumentation dialogue* [21] and transmitted via the chatbot.

IV. Results and Discussion

A microservice architecture brings a number of benefits, including *technological heterogeneity*, *replaceability and composability*, *ease of deployment*, *resilience* and *scalability*. In the following sections (IV-A–IV-E), we explore each of these properties, and explain how during the development of CONSULT they allowed us to address the challenges highlighted with patient-facing research software in Section I. We also describe various limitations of the approach encountered, and the mitigation strategies put in place.

A. Technological heterogeneity

The multi-disciplinary sub-groups tasked with building the CONSULT DSS comprised researchers with backgrounds in infrastructure development, AI and/or medical statistics. As such, each had different preferences for the way software should be implemented, based on their own experience, their research goals and the type of functionality they were tasked with adding to the system. For example, it was deemed necessary to develop the argumentation engine (Fig. 1, green) in Python, predominantly due to the fact that several packages key to the construction of the engine were realised in this language; to develop the traditional DSS components, such as the coordination components (Fig. 1, yellow) in Javascript, to leverage its ability to scale natively; and to develop the data miner and UI backend (Fig. 1, orange) in R, owing to the statistical processing that would need to be performed on the incoming data and the requirement to display aggregate statistics and graphs, respectively. While a decision on a uniform technology could have been made, this would have resulted in difficult trade-offs in the research to be conducted.

Instead, by realising the different components in a system as *microservices*, each with its own REST interface uniformly accessible via HTTP (or a queue intermediary), the choice of technologies underpinning the services can be heterogeneous. In the case of CONSULT, this meant that each research sub-team was able to develop in any of these languages without affecting the ability of the services to combine in order to deliver system functionality. For example, the Expressjs[7], Fastify[8], Flask[9] and Plumber[10] web frameworks were used to develop services written in Javascript, Typescript, Python and R, respectively, each of which is deployed within (custom) Docker containers. Despite the range of languages used, from Fig. 2 we can see that the dialogue manager (Javascript) is readily able to communicate with the argumentation engine (Python).

Technological heterogeneity is arguably the most immediate benefit of adopting a microservice approach for the development of patient-facing software, but we observed that unchecked heterogeneity should be approached with caution. This was evident in our choice of R as the underlying language for the data miner and UI backend services, which, although performing sufficiently as a part of the platform, is not necessarily designed to act in a service capacity. As a result, a number of additional solutions were required to solve several problems, including the introduction of a custom DNS server in order to ensure that the R-based components were able to correctly resolve the addresses of other services.

B. Replaceability and Composability

Although the incorporation of an argumentation-based reasoning engine into the CONSULT system has provided us with a suitable basis for the patient-based research we wish to conduct, during the development of the platform it was also desirable to decouple this component from the rest of the system in order to enable additional research to take place in the future with different, perhaps domain-specific, reasoners. The same was true of other components of the system, such as the data miner. However, it was clear that building this flexibility into the system could not compromise its stability, as patient-facing software needs to provide a reliable experience

[7]https://expressjs.com
[8]https://fastify.io
[9]https://flask.palletsprojects.com
[10]https://rplumber.io

TABLE I: Extract of dialogue templating syntax

Term	Description	Example
Dialogue	Unique identifier for patient dialogue.	*2-1-initiate*
Step	A unique identifier for the chatbot response given at this stage of the dialogue.	*1011*
BranchLevel	An indication of how many times the dialogue has branched (i.e. how many responses have been received from the user prior to this response) at this stage of the dialogue.	*2* i.e. one previous response from a user.
External	Indicates that the response to the user is not hard coded, but collected from an external service.	-
URL	The address of the service from which to retrieve data for a user response.	*http://service/argengine/explanation*
Body	A collection of key value pairs, indicating the data fields to be sent to the external service and where to source the data itself.	-
Type	The type of data to collect to send to the service.	*context* (access data stored by the dialogue manager itself), *literal* (use a hard coded value) or *external* (call an additional service, such as the data miner, for embedded dynamic data).
Print	A regular expression indicating which field to print from the external service response.	*.*expl.** (any field in the response containing the substring 'expl').

to users. By adopting a microservice design approach, we were able to develop a system where components are encapsulated entities which sit behind interfaces that remain consistent in the event that the underlying logic changes, and can thus be readily changed. For example, the reasoner service offers an endpoint to which a symptom, such as back pain, can be passed, along with a set of patient data. In turn, this endpoint promises to return a set of possible treatments for the patient (e.g. the administration of a drug specific to that symptom). With this contract in place, the services reliant on the reasoner are agnostic to the logic that provides this functionality in practice, be this the computational argumentation engine or another AI component.

Despite the separation between interface and functionality brought by a service-based approach, it became clear during the development of CONSULT that due to the sheer diversity of AI tools that might be plugged into the system in the future, minor modifications to the reasoner's interface were still inevitable. In an attempt to mitigate the impact of this, we explicitly built support for different reasoners into CONSULT's dialogue manager service. This service controls the operation of the chatbot by, in part, interfacing with the chosen reasoner (Fig. 2). To do this, rather than hard-coding the interactions between the dialogue manager and the reasoner we instead introduced what we term a *dialogue templating syntax*[11].

```
"Dialogue": "2-1-initiate",
"Step": "1011",
"BranchLevel": "2",
"External": {
  "serviceURL": "http://service/argengine/explanation",
  "Method": "POST",
  "Body": [
    {
      "Key": "pid",
      "Value": {
        "Type": "context",
        "Key": "user"
      }
    ...
  "Print": ".*expl.*"
...
```

Fig. 4: Extract from CONSULT dialogue template

An overview of the syntax is given in Table I, with an example shown in Fig. 4. Here, an extract is shown from a larger JSON-based template that is being used to specify how the chatbot should initiate a conversation with a user when an exacerbation in their vital signs (e.g. blood pressure) is detected. At this point in the dialogue—which consists of a number of *steps* and *branches* to model the space of possible responses given and allow the dialogue manager to correctly orchestrate user interactions—a response from the reasoner is required in order to fulfil a patient's request for an *explanation* of a piece of advice that has been given. This is specified by providing the address of the reasoner (indicated by the *serviceURL* field), a copy of the data it requires (e.g. a patient's demographic information, such as their unique user id, as stored in a key called *user* by the dialogue manager, and to be passed in a field called *pid*) and an indication of how to interpret the response it provides (in this case a field returned from the argumentation engine containing the text *expl*). This data can then be collected by the dialogue manager, sent to the reasoner and a response returned to the user. Our syntax can also be used to dynamically create options for a user, based on the response from the reasoner. Using this approach, in the event that the endpoint of a service does need to change, it is much easier to update how dependent components (in this case the dialogue manager) interface with it, with no updates to the code itself required.

The REFLECT project. The flexibility and stability offered by a microservice architecture can also be demonstrated by adopting the reverse perspective, and considering not only how the paradigm enables components to be swapped into CONSULT, but also how it enables CONSULT's services to be used elsewhere. As an example of this, we consider the cardiometabolic DSS developed by the global health technology company Metadvice Ltd[12]. This DSS aims to support

[11] An early version of this syntax was originally demonstrated (but not documented) in [16].

[12] https://www.metadvice.com

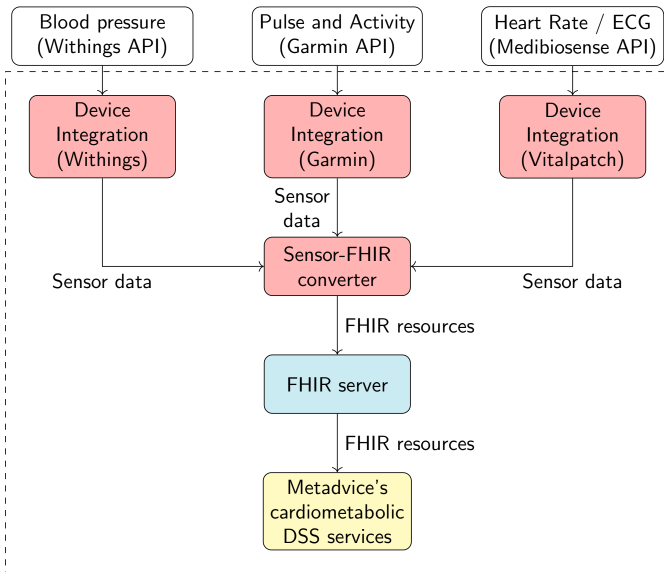


Fig. 5: The CONSULT services used to compose Metadvice's DSS

clinicians—and, as it evolves, patients directly—in the treatment of conditions such as hypertension and diabetes, and is composed of several services including a selection of those developed in CONSULT. The services reused from CONSULT include the wearable data device collection services and the FHIR server, shown in Fig. 5, which provide Metadvice's DSS with patient data that can then be displayed to a user and, ultimately, be used to support the treatment recommendations provided. Using a dedicated user interface, patients can, as they do in CONSULT, register their devices, with the other services in Metadvice's DSS then consuming the data collected. As such, this DSS can now be used in practice, with wearable device data providing a more holistic and real-time picture of a patient's health. The integration of CONSULT's components in this way was conducted as a part of the REFLECT project, which is designed to explore, more widely, the impact of wearable device data on AI-based decision-support systems, and thus on personalised patient healthcare.

C. Ease of deployment

A key component of a reproducible research method is ensuring the code or software used to derive a set of results is readily available and can easily be deployed in new environments. This acts as a record for how a set of results were derived and provides other researchers with the opportunity to replicate the results obtained. Microservice architectures are typically considered easy to deploy for a number of different reasons and thus have a positive impact on reproducibility. Primarily, the ability to deploy services independently of each other—and, critically, the ability to stop, fix and redeploy services without affecting the operation of other services in the event of issues with a new deployment environment—makes the technical deployment of a platform designed as a set of services more manageable. Moreover, the natural choice to containerise services eliminates many of the issues associated with directly installing a piece of software to a new environment, including package installation overhead (with these already being stored within an image or automatically installed when building an image) and the overhead of deploying software to remote locations (e.g. the use of image registries to deploy services to a remote server efficiently). At a high level, the separation of a system into individual services, each with a clear remit, also acts as a form of documentation for those wishing to gain an overview of how a system works.

For patient-facing software, having a platform that can be deployed in this manner is particularly beneficial in respect of reproducibility. When a piece of software is designed to be used by patients, there are elements of the logic of that system which it may not be possible to make public, for security and/or privacy reasons. This is likely to include those elements that deal with the processing and storage of patient data. Within a monolithic system, the absence of a component like this would likely prohibit deployment, and, as a result, limit reproducibility. However, the ability to deploy services separately, as brought by a microservice architecture, means that in the absence of certain services parts of the architecture, not connected to the missing services, can still be deployed and run and their outputs verified. Moreover, those services that are not available can easily be substituted with different services to form a complete architecture. Within CONSULT a component with limited public availability is the FHIR health data server, which has been customised for our use (Fig. 1). Because of the service structure, the absence of this data server does not preclude, for example, another researcher running the data miner, and verifying its operation with other patient data or with synthetic data. In addition, one could easily substitute the missing data server with a generic FHIR server, in order to have a complete running system.

Despite these benefits, if an appropriate number of services are not used, this structure can, instead, have a negative impact on ease of deployment, and thus reproducibility. During the lifetime of the CONSULT project, we have transitioned between different phases of the study (e.g. pilot to user), and the system development has changed hands between different groups of researchers, with new researchers often needing to run the platform in new environments, and indeed wanting to verify the operation of the system before proceeding with the next phase of the work. Here, we have seen in practice a number of the stated deployment benefits of having an architecture that comprises a number of services. For example, looking at the architecture of CONSULT (Fig. 1) one can gain an initial insight into how its components combine to deliver the functionality of the platform. At the same time, some researchers have reported that the number of services in the system seems to outweigh the intelligibility brought by this separation, obfuscating their ability to understand and deploy it. As such, while it is clear that having services with a limited remit is often a positive in terms of conveying the functionality of the system to new developers, having too many services can result in a system that is unwieldy. Thus, an appropriate

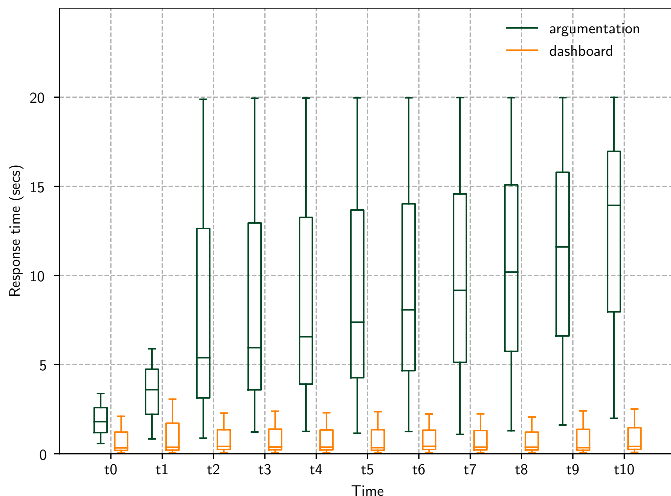


Fig. 6: How CONSULT's dashboard service responds to increased load on the argumentation service

balance needs to be found, which may vary from project to project.

D. Resilience

Although we have seen that technological heterogeneity allows various components to communicate with each other, the ability of each component to engage in said communication may still be limited. One of these limiting factors is the time taken for a component to complete its task when the system receives a given input. During the development of CONSULT, the required computational argumentation functionality—which drives the AI research around which the project's primary contributions are centred—was identified as an element of the proposed system likely to be associated with higher processing times, even assuming a (Python) implementation that is optimised to make maximum use of the available processing cores on a given machine. As such, its execution risked affecting the ability of other areas of the system not reliant on its outputs, such as the aggregation and presentation of sensor data, to respond to users in a timely manner.

By separating the individual components of the system into self-contained services, we gained a number of benefits in respect of processing time. Firstly, like the health data server, this allowed us to place the computational argumentation service on a dedicated virtual machine, to which we can allocate additional resources in an attempt to mitigate any delays in advance. In addition, assuming there are still delays introduced by the argumentation components, we were able to ensure that, within CONSULT, these delays do not affect users' access to components unrelated to the engine. A user can, for example, still interact with the dashboard to load an aggregated view of their sensor data from the server, as long processing times within the argumentation component do not propagate to the rest of the system as they would in a typical monolithic structure.

In other words, a microservice architecture provides us with a high level of resilience. We are able to confirm this effect by sending controlled amounts of HTTP load to both the argumentation service and the dashboard (UI backend) service simultaneously. These services are deployed as detailed in Section III. Our aim is to identify any dependencies between the services by comparing trends in their respective response times (as opposed to comparing their relative performance, as this would not be a fair test given that the services differ in terms of functionality, inputs and execution environment). To ensure uniformity in all other aspects of the tests, we use a standard load testing tool[13]. Note that no automated service scaling techniques are employed for these tests. From Fig. 6, we can see that if we increase the load by 50 concurrent requests/sec at t_n $(0 < n \leq 10)$ for the argumentation service, its response time degrades, however the response time of the dashboard, which we keep under consistent load (50 concurrent requests/sec), does not exhibit such a trend. To quantify this further, using Ordinary Least-Squares (OLS) regression we can estimate[14] the dependency between response time and increased load in our tests. For the argumentation service, OLS gives us a coefficient of 0.1 $(p < 0.05)$ suggesting a positive relationship between response time and load, whereas a much lower (< 0.01), statistically insignificant coefficient is received for the dashboard, suggesting it is not impacted by the increased load on the argumentation service.

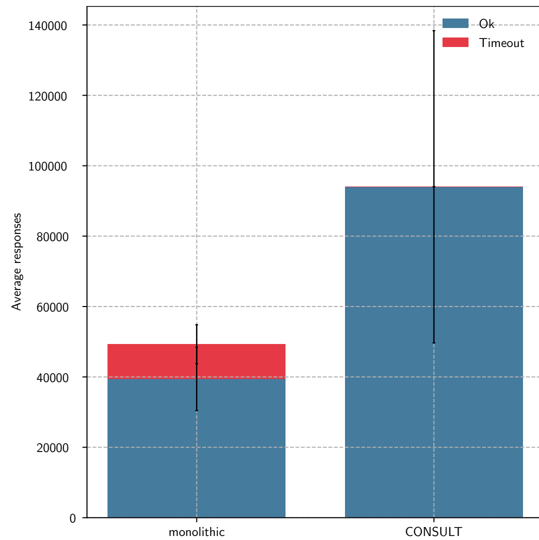
Naturally, for those components that are reliant on outputs from the argumentation engine, any delay does have an impact. For example, the chatbot will still have to wait for outputs from the argumentation engine before responding to a user, e.g. in the scenario shown in Fig. 2. However, because the chatbot is simply waiting, as opposed to being delayed by processing capacity, there is still the capacity to respond to a user, to inform them, for example, of longer running times, while the argumentation engine is still processing.

E. Scalability

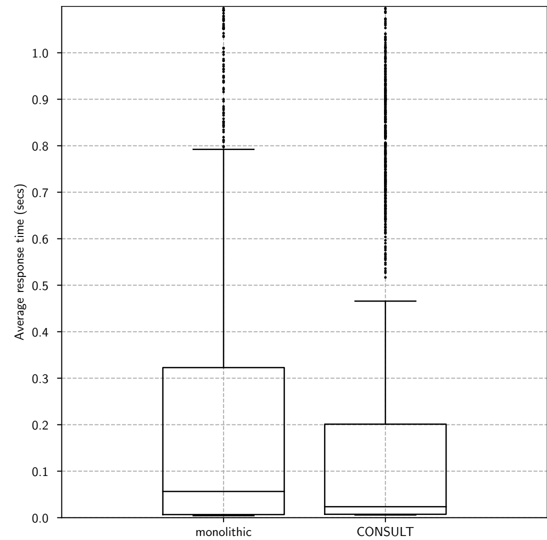
The data gathered from wearable devices by the CONSULT system—and used as a key input to the argumentation engine—can be characterised by a number of properties. Predominantly, the devices of interest have high-volume data output, in particular the target ECG patch, which takes multiple readings every second over a continuous period of up to 7 days. The amount of data from some of these devices can, however, vary, depending on the number of devices uploading data at any given time; in some cases there can be a large amount of data being passed to the system, and in others none at all. Finally, the mechanism offered by each vendor to supply the data in practice can differ, with some offering push-based models, and others requiring polling for available data. Given each of these properties, there was a risk that the system developed would not be able to collect all the available

[13]https://github.com/rakyll/hey

[14]Due to the autocorrelation typically found in time series data, we only treat the derived coefficients as estimates.



(a) Number of requests responded to and portion of timeouts



(b) Response time

Fig. 7: How CONSULT responds to requests at high load compared to a monolithic system

data, leading to an incomplete and thus inaccurate view of each patient.

The adoption of a microservice design approach allowed us to ensure that all the data available to CONSULT is adequately collected. Firstly, by using the principle of limited service scope as a guide, we were prompted to develop a different service for each vendor (Withings, Garmin and Medibiosense, Fig. 1, red), ensuring that, from an integration perspective, data could be collected sufficiently. Each service is able to collect data based upon that vendor's configuration: receive, and subsequently poll for, new data in the case of Withings; receive a direct push of data in the case of Garmin; and continuously poll for data in the case of Medibiosense. In addition, to address the varying volume of data, the microservice approach provides us with the ability to selectively scale these services, both up and down, depending on demand (requests per second (RPS)). In practice, we achieve this scalability using our container orchestration tooling. This complements the inherent scalability provided by building these services using a language like V8-compiled Javascript, which prioritises scalability by design. With the services appropriately scaled based on load, we remove the risk of not being able to capture all the data passed to the system.

We can confirm the scalability of our architecture in practice by once again issuing a controlled amount of load to the system, this time targeting the Withings device integration service. For comparison, we emulated a monolithic system that does not scale by placing a single replica of this service's logic (including a gateway) on a single node. For CONSULT, RPS auto-scaling was enabled (up to 20 service replicas across three nodes). Requests were issued to both systems in repeated 60s windows. All other test conditions were as described in Section IV-D. Across these two systems we then determined: (1) the average number of requests that can be responded to, and, of those requests, which are positive responses and which indicate a load-based error; and (2) the average time taken to respond to those requests. From Fig. 7a, we are able to confirm that, despite quite a large standard deviation in the average number of responses from CONSULT, when services can be replicated according to demand, a much higher number of requests can be responded to. More importantly, we can see that in the monolithic system a number of requests time out, meaning the data contained within those requests (e.g. pushed heart rate data) is lost. In contrast, virtually no timeouts are found in CONSULT. Similarly, from Fig. 7b, we can see that the average response time is lower from CONSULT; despite the overlapping interquartile ranges (IQRs), a Mann-Whitney U-test, under the null hypothesis that there is no difference between the two distributions, shows a significant difference between the two average response times ($p < 0.05$). We attribute this to the fact that while a monolithic system is processing existing requests, new requests are effectively held in the gateway until they can be responded to, leading to an increased overall response time and, in certain cases, the timeouts seen.

While the separation of services in a microservice architecture provides us with many benefits, the fact that data is now passed between a number of different components for processing creates a new point of failure. Specifically, if, upon the receipt of an HTTP request from a service, the receiver throws an error, the information sent is lost (assuming no replay mechanism is present). If this occurs when processing collected data, scaling up the services in our system will not prevent an incomplete picture of a patient being received. Our use of messaging queues (Fig. 2), where sensor data is placed after being collected by our device integration services (and where patient records could be placed by our EHR integration service) rather than being sent over HTTP, ensures that data

is not only completely captured from a source but also stored in its entirety. Each queue operates using the AMQP protocol (as provided by the RabbitMQ message broker[15]). With this in place, sensor data (and EHR data) remains present even in the event of issues with the services at each end of the queue.

There are a number of other areas of the system that also benefit from the scalability of services. These include the dashboard, where each user interacts with a replica of the backend service, ensuring maximum availability. However scaling at the service level is unlikely to have an impact in certain areas of our architecture, such as the reasoner, where, although desirable, it may not be trivial to split tasks between different replicas in order to reduce processing time.

V. Summary and Conclusion

In this work, we have shown how the requirements of researchers often place at risk the usability of software in patient-based research. Researchers want to (i) use programming languages that are geared more towards experimentation and data analysis than scalability; (ii) change the components of a system frequently and experiment with components that have high execution times; (iii) use potentially intermittent and complex data sources; and (iv) share the software used to derive their results. To ensure that research software that is developed and used in this manner is suitable for patient-based research we have proposed the use of a microservice design approach. This approach has a number of advantages: (i) different languages—suited both to data analysis and the development of user software—can be mixed; (ii) components can be changed and components with high processing requirements can be used without sacrificing system stability or response time; (iii) a variety of different data sources can be used without losing data important for a system to compile an accurate view of a patient; and (iv) software can be shared as part of a reproducible research methodology without compromising security. We have shown these advantages in practice, and how the approach can reconcile the needs of both researchers and patients, by introducing the CONSULT system, how it was designed and how it performs under load. It is worth noting that the connections we have drawn between each of the benefits brought by the use of a microservice architecture and a given problem in the development of patient-facing research software are not designed to be exclusive. For example, composability, while providing flexibility, also has a significant impact on reproducibility. Moreover, it is not our claim that monolithic, patient-facing research software can never exhibit these quality attributes; a well-designed, well-resourced monolithic application may, for example, exhibit some resilience. However, we believe that by building an application as a set of microservices these properties are much easier to obtain.

As well as highlighting the benefits of a microservice approach, we have also commented on its limitations in respect of reconciling the needs of patients and researchers. In turn, we have suggested potential mitigation strategies, in particular the use of a dialogue templating syntax to deal with inevitable changes to service interfaces. However, despite our mitigations, it is also clear that microservices do not in themselves guarantee software that is perfectly suited to patient use. The adoption of a particular architecture has no impact on, for example, the quality of the UI served by a UI service and whether that UI is suited to patients who may be impaired, either cognitively or physically.

Future work will explore the impact of the microservice approach on the development and performance of different types of patient-facing research software. With patients taking greater responsibility for their own care, this is likely to include research software that is not necessarily designed to be used by patients but has an increasing patient-based audience. EHR-based phenotype libraries, for example, now make public the definitions used to identify medical conditions in research studies, and are thus likely to also attract interest from patients; something that should be factored into their design. We have demonstrated the benefits of using microservices in the phenomics domain through our work with the Phenoflow library which uses microservices to store, generate and visualise computable phenotypes for researchers [22]. Looking further ahead still, the benefits of using a microservice design approach likely extends to other groups of end-users and thus to other types of research software, and this is worthy of additional investigation. The most immediate group of users to consider would be clinicians, where microservices could, for example, be used to modularise EHR system architectures to obtain similar benefits to the ones seen, such as the ability to have custom, stable deployments in different health settings.

[15]https://rabbitmq.com

References

[1] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly, 2019.

[2] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[3] A. Blohowiak, A. Basiri, L. Hochstein, and C. Rosenthal, "A platform for automating chaos experiments," in *Proceedings of the 27th International Symposium on Software Reliability Engineering (ISSREW 2016)*, 2016, pp. 5–8.

[4] K. Lillaney, D. Kleissas, A. Eusman, E. Perlman, W. Gray Roncal, J. T. Vogelstein, and R. Burns, "Building NDStore through hierarchical storage management and microservice processing," in *Proceedings of the IEEE 14th International Conference on e-Science (e-Science)*, 2018, pp. 223–233.

[5] A. Johanson, S. Flögel, C. Dullo, and W. Hasselbring, "OceanTEA: Exploring ocean-derived climate data using microservices," in *Proceedings of the 6th International Workshop on Climate Informatics (CI 2016)*, 2016, pp. 25–28.

[6] N. T. De Sousa, W. Hasselbring, T. Weber, and D. Kranzlmüller, "Designing a generic research data infrastructure architecture with continuous software engineering," in *Proceedings of the 3rd Workshop on Continuous Software Engineering (CSE 2018)*, 2018, pp. 85–88.

[7] C. Schröer, "Towards microservice identification approaches for architecting data science workflows," in *Proceedings of the International Conference on Health and Social Care Information Systems and Technologies (HCist 2021)*, 2021, pp. 519–525.

[8] F. M. Garcia-Moreno, M. Bermudez-Edo, J. L. Garrido, E. Rodríguez-García, J. Manuel Pérez-Mármol, and M. José Rodríguez-Fórtiz, "A Microservices e-Health System for Ecological Frailty Assessment Using Wearables," *Sensors*, vol. 20, no. 12, pp. 1–23, 2020.

[9] S. Roca, J. Sancho, J. García, and Á. Alesanco, “Microservice chatbot architecture for chronic patient support,” *Journal of Biomedical Informatics*, vol. 102, no. 2, pp. 1–9, 2020.

[10] B. C. Delaney, V. Curcin, A. Andreasson, T. N. Arvanitis, H. Bastiaens, D. Corrigan, J.-F. Ethier, O. Kostopoulou, W. Kuchinke, M. McGilchrist, P. van Royen, and P. Wagner, “Translational Medicine and Patient Safety in Europe: TRANSFoRm—Architecture for the Learning Health System in Europe,” *BioMed Research International*, vol. 2015, pp. 1–8, 2015.

[11] J. S. Hirsch, J. S. Tanenbaum, S. L. Gorman, C. Liu, E. Schmitz, D. Hashorva, A. Ervits, D. Vawdrey, M. Sturm, and N. Elhadad, “HARVEST, a longitudinal patient record summarizer,” *Journal of the American Medical Informatics Association*, vol. 22, no. 2, pp. 263—274, 2015.

[12] N. Kökciyan, M. Chapman, P. Balatsoukas, I. Sassoon, K. Essers, M. Ashworth, V. Curcin, S. Modgil, S. Parsons, and E. Sklar, “A collaborative decision support tool for managing chronic conditions,” in *Proceedings of the 17th World Congress of Medical and Health Informatics (MedInfo 2019)*, vol. 264, 2019, pp. 644–648.

[13] I. Rahwan and G. R. Simari, Eds., *Argumentation in Artificial Intelligence*. Springer-Verlag, 2009.

[14] N. Kökciyan, I. Sassoon, E. Sklar, S. Modgil, and S. Parsons, “Applying metalevel argumentation frameworks to support medical decision making,” *IEEE Intelligent Systems*, vol. 36, no. 2, pp. 64–71, 2021.

[15] N. Kökciyan, S. Parsons, I. Sassoon, E. Sklar, and S. Modgil, “An argumentation-based approach for generating domain-specific explanations,” in *Proceedings of the 17th European Conference on Multi-Agent Systems (EUMAS 2020)*, 2020, pp. 319–337.

[16] K. Essers, M. Chapman, N. Kökciyan, I. Sassoon, T. Porat, P. Balatsoukas, P. Young, M. Ashworth, V. Curcin, S. M. l, S. Parsons, and E. I. Sklar, “The CONSULT System: Demonstration,” in *Proceedings of the 6th International Conference on Human-Agent Interaction (HAI 2018)*, 2018.

[17] M. Chapman, P. Balatsoukas, M. Ashworth, V. Curcin, N. Kökciyan, K. Essers, I. Sassoon, S. Modgil, S. Parsons, and E. I. Sklar, “Computational Argumentation-based Clinical Decision Support,” in *Proceedings of the 18th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2019)*, 2019.

[18] P. Balatsoukas, I. Sassoon, M. Chapman, N. Kökciyan, A. Drake, S. Modgil, M. Ashworth, V. Curcin, E. Sklar, and S. Parsons, “In the wild usability assessment of a connected health system for stroke self management: a pilot,” in *Proceedings of the 8th IEEE International Conference on Healthcare Informatics (ICHI 2020)*, 2020, pp. 456–458.

[19] D. Bender and K. Sartipi, “HL7 FHIR: An agile and RESTful approach to healthcare information exchange,” in *Proceedings of the 26th IEEE International Symposium on Computer-Based Medical Systems (CBMS 2013)*, 2013, pp. 326–331.

[20] V. Vernon, *Implementing domain-driven design*. Addison-Wesley, 2013.

[21] D. Walton and E. C. W. Krabbe, *Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning*. State University of New York Press, 1995.

[22] M. Chapman, L. Rasmussen, J. Pacheco, and V. Curcin, “Phenoflow: A Microservice Architecture for Portable Workflow-based Phenotype Definitions,” in *Proceedings of AMIA Joint Summits on Translational Science*, 2021, pp. 142–151.