Imperial College London

Imperial College London

Department of Bioengineering

M.Eng. Individual Project

Final Report

ULAS: A Universal Lab Automation System for Synthetic Biology

---

Author:

Yong Shen Tan
(CID: 01055349)

Supervised By:

Dr. Guy-Bart Stan
Dr. Zoltan A. Tuza

June 16, 2021

Word Count: 5990

Submitted in partial fulfilment of the requirements for the award of
M.Eng. in Biomedical Engineering from Imperial College London

Contents

List of Figures

List of Tables

1 Abstract

With the rise of technology, automation has become pivotal to enhancing many facets of our lives – from manufacturing to logistics, IT to smart home technology, and the list goes on. However, the potential for automation to improve the way scientific research is conducted in the lab has largely been left untapped due to a myriad of reasons such as prohibitive costs, fear of vendor lock-in, as well as lack of interoperability between different systems.

This paper proposes a universal lab automation system, named ULAS, that enables one to build an interconnected laboratory of the future by integrating instruments and tools within the lab into a centralised software platform. Through this platform, one can then leverage these integrations to design and orchestrate the automation of experimental workflows in a data-centric way.

By establishing standardised interfaces and communication protocols, ULAS was designed with extensibility in mind such that it can be tailored to the needs of each lab. Through open-source, the project hopes to provide a framework on which other labs as well as device manufacturers can build upon to create a robust, cost-effective and open ecosystem that can be easily adopted for research.

Finally, this paper goes on to further demonstrate how ULAS can be harnessed to automate some common experimental protocols found in the field of synthetic biology research, with the ultimate goal of illustrating how such a platform can help improve the reproducibility and scalability of research, whilst also easing the cognitive overhead faced by researchers.

2 Acknowledgements

I would first like to express my deepest gratitude to my project supervisors, Dr. Guy-Bart Stan and Dr. Zoltan Tuza, for their utmost support and guidance throughout the course of the project.

Additionally, I would also like to thank the research members of the Stan Group from the Imperial College Centre for Synthetic Biology for their expert advice and feedback that has helped shape the outcome of the project.

Finally, I wish to also show my appreciation towards Prof. Harrison Steel (creator of the Chi.Bio), as well as the staff members at Riffyn, Inc. and Tecan Group Ltd., for the technical support that they have provided in utilising their products.

3 Introduction

3.1 Motivation

3.1.1 The Case for Laboratory Automation

Despite reproducibility being a cornerstone of scientific research, it is often an uphill challenge to achieve, with more than 70% of researchers surveyed failing to reproduce results found in literature as well as at least 50% unable to reproduce their own [3]. On top of this, this reproducibility crisis often comes with a huge economic toll - in the United States alone, the cost of irreproducibility in preclinical research comes up to at least $28 billion annually [4]. Ill-defined experimental protocols, inconsistent use of lab instruments, and the susceptibility of experimentation to human error are factors that contribute to the pervasiveness of this problem.

The scalability of research is also another challenge facing scientists. A recent white paper published by Synthace claimed that despite huge technological advancements, a majority of biological research is still conducted in a manual and reductionist manner that makes it difficult to scale beyond controlled environments [5]. The repetitive and involved nature of research means that this is especially true for labs that lack the capacity to invest in sophisticated equipment, such as those used in the pharmaceutical industry. Scalability is essential for achieving high-throughput research and accelerating the quest for scientific breakthroughs as highlighted by the rapid development of COVID-19 vaccines [6].

Add on the vast and disparate ecosystem of tools at researchers' disposal, laboratory automation is often seen by the scientific community as the answer to the above problems. The laboratory of the future is an interconnected one that leverages the integration of devices and data within the lab in a synergistic and cost-effective manner. This can be achieved through adopting open software and the Internet of Things (IoT), as well as standardising integration interfaces and communication protocols between devices [7].

Such an interconnected ecosystem not only enables researchers to define their experimental protocols precisely and unambiguously through software, but it also abstracts away the error-prone and toil-heavy nature of research by relying on machine-executed experiments that can even be performed remotely. Moreover, lab automation that leverages the ubiquitous nature of the Cloud means that researchers can easily peer-review and collaborate on experimental protocols and data.

3.1.2 Obstacles in Laboratory Automation

Although lab automation is seen as a holy grail for research, adopting it often comes with its own set of challenges. While it is true that modern equipment like liquid handling robots and plate readers are commonplace in most labs today, the potential of lab automation is severely hindered by the lack of interoperability between these devices. In a world where the movement of data is so crucial to enabling seamless and consistent experiences, such as those made possible by Open Banking, the lack of a publicly recognised standard for device integration means that it can be extremely difficult to assimilate equipment from different vendors into a unified framework [8]. Add on to that the wildly different data formats that are used, reconciling experimental data is often another huge challenge in itself [9].

Proprietary and legacy systems add another dimension to this interoperability problem as they can be impossible or extremely difficult to integrate with [10]. Some workarounds involve low-level manipulation or modifying source code, but these methods are often fragile and prone to breaking when interfaces change. Given that even basic experiments require the orchestration of multiple instruments, this interoperability causes information disjoints that can sometimes incur additional effort on researchers to reconcile, severely hindering the appeal of lab automation [11].

To make matters worse, the fear of vendor lock-in is a very practical concern that can make labs think twice before investing in automation - settling on a solution that's only compatible with equipment from specific vendors could impact their mobility to upgrade them to better alternatives in the future. Thus, labs are wary of adopting solutions that lack extensibility and are not vendor-agnostic.

Finally, Bär et al. discuss other prominent factors contributing to the low adoption rate of lab automation solutions, asserting that the "high cost, the long implementation times, and the inflexibility of such systems restrict the penetration into smaller laboratories in academia" [8]. In addition, these solutions typically require specialised knowledge to operate and maintain. Inertia to change also cripples the willingness of lab managers to invest in IT infrastructure and researchers to learn and adapt to new technologies and practices in their routines [12].

3.1.3 Existing Literature and Solutions

To help address the problem of interoperability and unlock the potential for automation, a growing number of solutions have arisen in recent years attempting to promote open connectivity and software-driven approaches to more seamless integration of devices and data. This is especially relevant given the rise of low-cost, highly-available IoT devices and equipment manufacturers picking up on the practice of exposing application programming interfaces (APIs) in their products. These existing literature and solutions take advantage of these to champion for encoding protocols through software as well as open standards for facilitating the integration of instruments and tools to enable automation [13].

However, a common pitfall for many of these solutions is that they tend to introduce new tools that researchers have to adapt into their workflows, rather than leverage ones that they already use. Their steep learning curves further raise their barriers to adoption. In some cases, they also tend to overlap with the roles played by laboratory information management systems (LIMSs) that have become essential to the lab environment.

Appendix A contains additional evaluation of some of the more prominent lab automation solutions that are publicly available, highlighting some of their main advantages and disadvantages.

3.2 Aims and Objectives

This paper proposes a universal lab automation system, named ULAS, that aims to allow one to build an interconnected laboratory of the future by integrating the instruments and tools within one's lab environment into a centralised software platform. Through this platform, one can then leverage these integrations in a synergistic and composable way to design and orchestrate the automation of experimental workflows, with protocols defined in a consistent and repeatable manner. Additionally, it also helps one to realise data-centric and closed-loop automation by integrating data across devices into workflows.

ULAS was also developed with extensibility in mind and thus seeks to enable labs to tailor it to their specific needs in a cost-effective manner. Although the use cases for automation covered in this project are largely specific to the Imperial College London Centre for Synthetic Biology, it provides a framework and foundation for others to adapt and build upon. Through open-source, this project also hopes to encourage those with vested interest, such as device manufacturers, to contribute plugins and integrations to ULAS that can be used by the community in a plug-and-play fashion.

Finally, this paper also seeks to demonstrate how ULAS can be harnessed to automate some common experimental protocols in the field of synthetic biology, which typically involves a Design-Build-Test-Learn cycle (Figure 1) that encompasses numerous experimental protocols, many of which can be repetitive and tedious in nature whilst also requiring a high degree of reproducibility.

ULAS helps tighten this feedback loop and address these issues of reproducibility and scalability through automation, increased standardisation and quality control, as well as reducing the cognitive overhead faced by researchers.

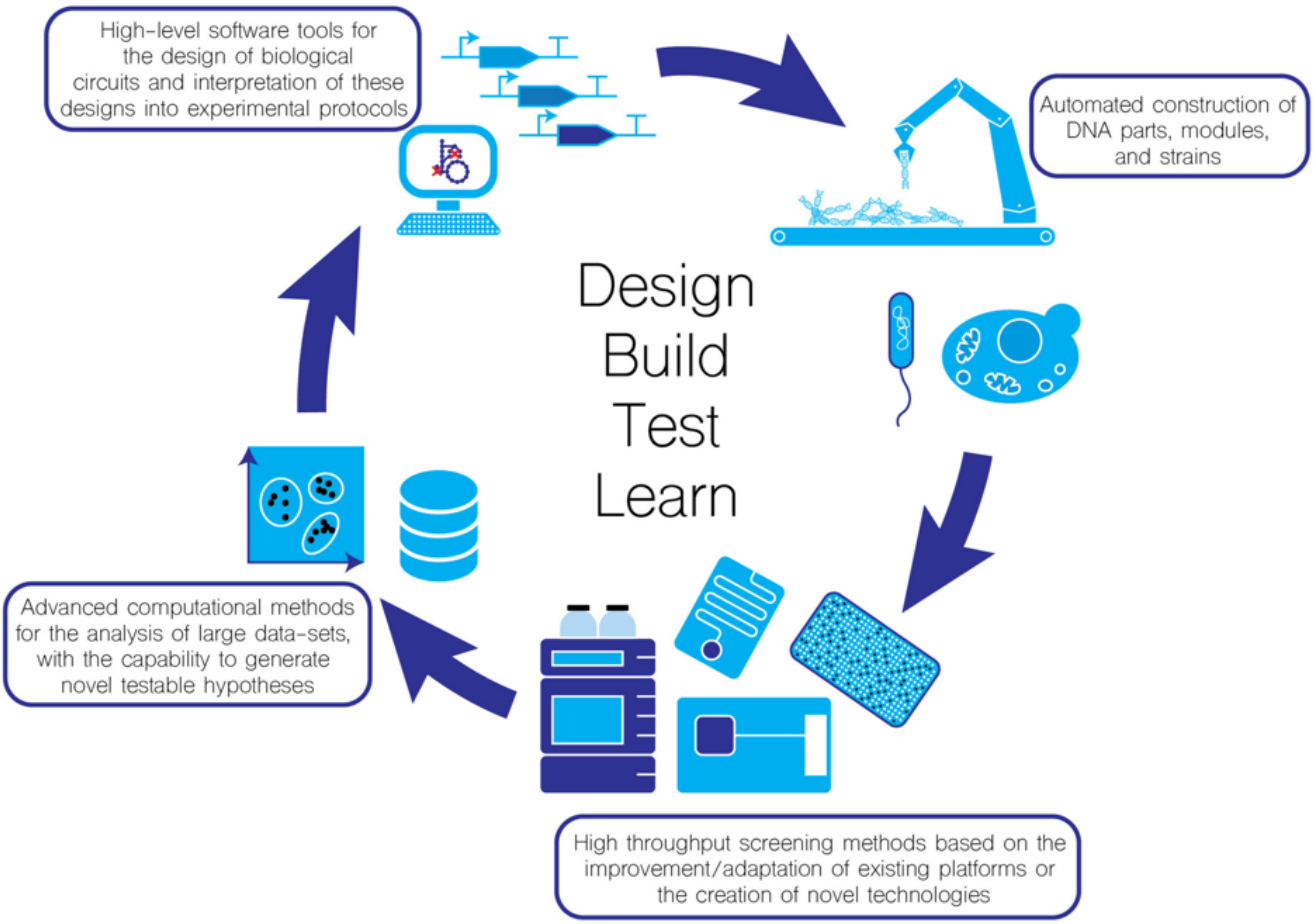

Figure 1: Design-Build-Test-Learn Cycle in Synthetic Biology [1]

The following project objectives have been defined to help achieve the aforementioned aims:

1. Design a modular and scalable software architecture for unifying devices into a standardised platform.
2. Design a robust and intuitive language specification for defining experimental protocols that can be used to power automation.
3. Identify several devices as targets for automation and develop the ULAS-compatible integrations for them.
4. Develop or integrate with a suitable user interface for researchers to manage the lifecycles of their automation workflows, from design to execution to data collection.
5. Automate some common experimental protocols used by researchers in the lab and benchmark them against existing processes.
6. Open-source the project and provide the necessary framework and documentation that the broader community can refer to for extending the platform.

4 Methods

4.1 Integration Targets

As the scope for laboratory automation has the potential to be limitless, the first step was to narrow down the integration targets that will make up the initial components of the ULAS platform. Amongst the devices available within the Imperial College Centre for Synthetic Biology, Table 1 details the set of automation targets - devices that perform the automation of experimental protocols - to be integrated with.

Device	Brief Overview
Chi.Bio	An open-source platform combining the capabilities of an incubator, shaker, plate reader, and liquid handler into a single unit.
Tecan Spark®	A multimode microplate reader intended to enhance and streamline biochemical and cell-based workflows.
Opentrons OT-2	An open-source and high precision liquid-handling robot.

Table 1: Overview of Selected Automation Targets

Additionally, Riffyn Nexus, a cloud-based process data system, was also chosen as an integration target albeit in a different role in the overall architecture. Through its web-based graphical user interface (GUI) for designing processes and recording of associated data, it serves as the ideal entrypoint for researchers to manage the lifecycles of their experiments automated by ULAS. It also allows one to "start" and "stop" *Experiment Runs*, as indicated by the red box in Figure 2, which can be used as events for triggering automation targets. In theory, other similar software tools could be used in place of Riffyn Nexus (covered in Section 4.2.1: Management Layer).

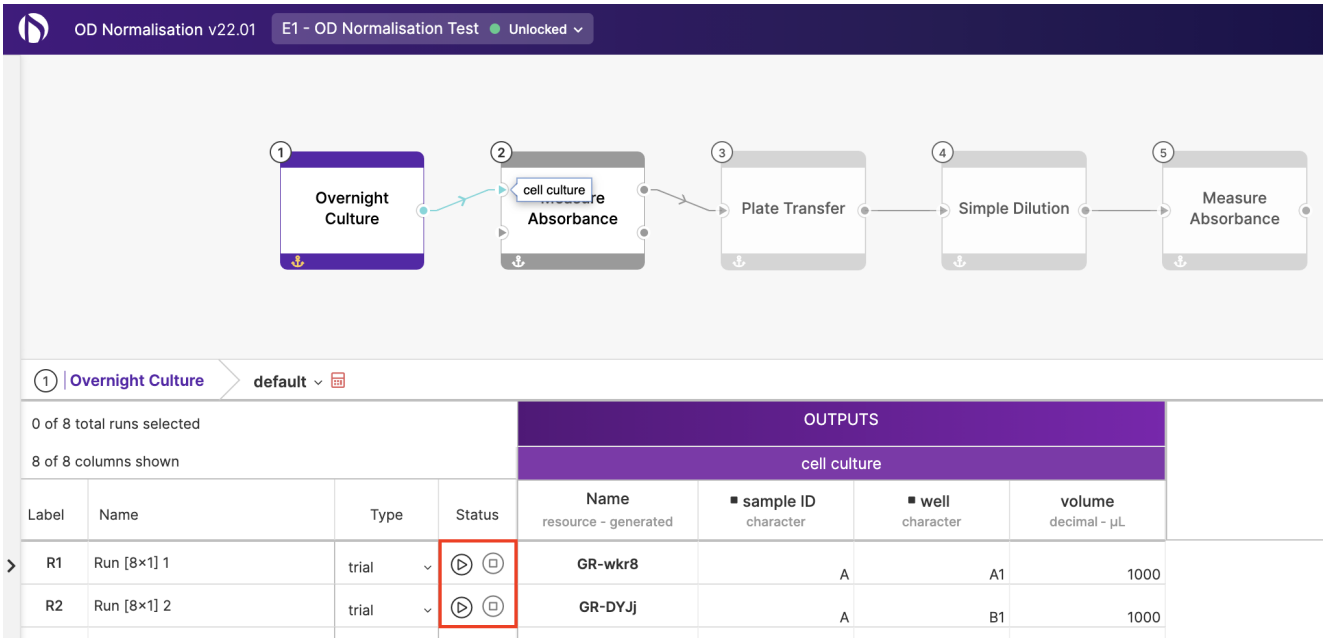

Figure 2: Example *Experiment* in Riffyn Nexus

The aforementioned integration targets were primarily chosen based on the following reasons:

- They are regularly used by researchers within the lab and are also well-recognised amongst the research community. For instance, Riffyn Nexus was vital for scaling up COVID-19 diagnostics in London [14].

- They typically require several manual steps to configure or utilise, which can be time-consuming and error-prone.
- The data they produce tend to be disconnected from the experimental source and hence requires additional processing to reconcile.
- They expose well-defined APIs, facilitating the development and testing of integration code.

In essence, the implementation for ULAS should provide a general framework that can act as the glue between Riffyn Nexus and the automation targets. One should be able to trigger the automation of an experimental protocol on a desired target device via Riffyn Nexus, with associated data being written back to the Riffyn Nexus database. This data can then be propagated and used in downstream steps of the experiment, providing seamless integration of data and closed-loop feedback across the entire workflow as illustrated by Figure 3. One can then use the data stored in Riffyn Nexus for further analysis.

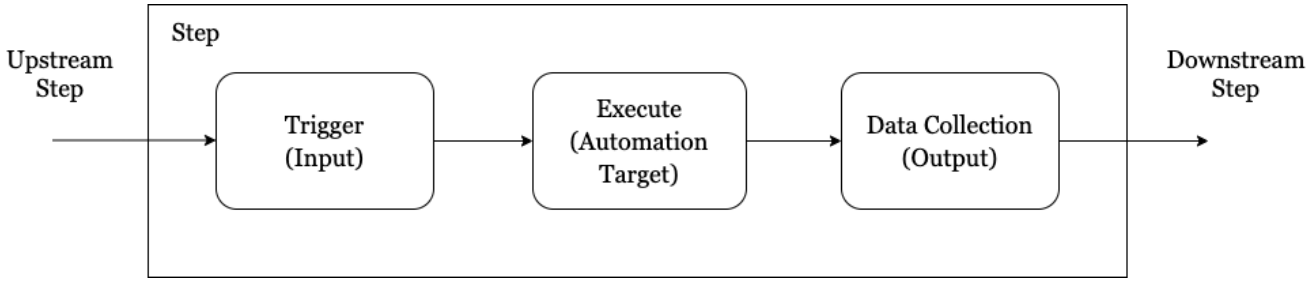

Figure 3: Outline of Step in Automation Workflow

4.2 Software Architecture

Given that the chosen integrations might not apply to every lab, the software architecture for ULAS needs to be highly modular and scalable such that components can be easily substituted or added. To satisfy this requirement, ULAS adopts a microservices-oriented architecture (Figure 4), where services are distributed across hosts and have isolated domains of responsibility, thus introducing a high degree of decoupling.

Unlike its monolithic counterpart, the decoupled nature of microservices allows one to easily replace a service without having to modify others, as long as interfaces remain unchanged. It also improves fault-tolerance such that a misbehaving service only impacts a restricted subset of features or services, rather than the entire system [15]. Finally, this architecture also facilitates the development and testing of integrations in isolation, which is especially useful when hardware dependencies are involved.

Most of the services in this project were written in Python, which was chosen as it is a programming language most researchers are familiar with, which could hopefully encourage them to contribute to the project. However, given the microservices architecture, services can technically be written in any language as long as they respect the expected interfaces, which is especially useful given the wide variations in interfaces exposed by integration targets.

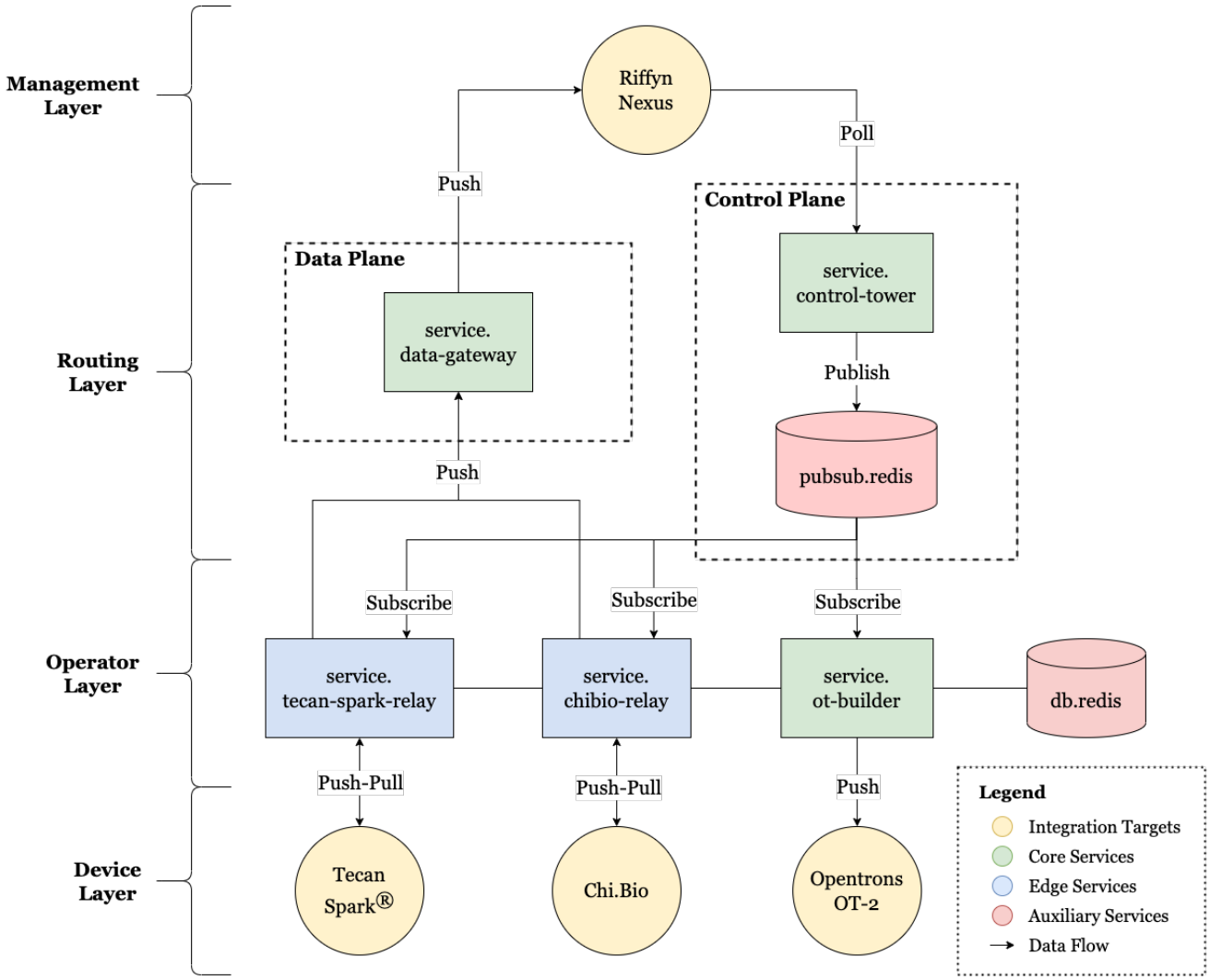

Figure 4: Microservices-Oriented Software Architecture of ULAS, Comprising of 4 Layers

In the distributed context, services can be classified into either Core or Edge - the former can run anywhere in the Cloud or on-premise, while the latter typically require access to physical hardware, local systems or APIs, and are thus required to be ran on peripheral hosts alongside the devices they are interfaced with. There also exists auxiliary services that play complementary roles such as databases, caches, message queues, etc.

To work in tandem, the services interact and exchange data by communicating over the network; most of the communication occurs over Hypertext Transfer Protocol (HTTP), a synchronous and stateless communication protocol. It was chosen as it is established and reliable, forming the foundation of most data exchanges on the Web. However, communication between the Control Plane and Operator Layer occurs asynchronously (see Section 4.2.2).

Most of these services run in the backend and do not require any user intervention or interaction - researchers can continue to work directly with the integrated devices albeit in a more automated fashion. In the case a service requires additional input from users, it can expose a web-based frontend. The software architecture underpinning ULAS can be further broken down into 4 distinct layers:

4.2.1 Management Layer

The Management Layer is the user-facing layer comprising of external software to be integrated with for allowing users to manage the lifecycles of automated workflows. Typical examples that fit well into this layer are information or process management systems that provide powerful user

interfaces for designing experimental workflows and capturing experimental data, such as Riffyn Nexus and Benchling. Additionally, multiple software can be used in the Management Layer simultaneously.

While one would typically use the same integration for both designing workflows and capturing data, it is not strictly necessitated by ULAS's design. It is entirely feasible to have one software for the former (eg. Benchling) and another for the latter (eg. Google Sheets). This, however, makes it more difficult to build closed-loop workflows where data from upstream steps are connected to downstream steps. Riffyn Nexus, being able to fulfil both these roles, while also providing a comprehensive GUI for data-centric processes, made it the perfect candidate for the Management Layer in this project.

One can also integrate their own custom software as long as it exposes an API that can be used to develop plugins for the Routing Layer.

4.2.2 Routing Layer

The Routing Layer is responsible for moving data around the system to the right locations. It is made up of the Control and Data Planes, which provide pluggable architectures for interfacing with the Management Layer.

Control Plane: The Control Plane is responsible for watching for events in the Management Layer and based on those events, issue messages to the relevant service in the Operator Layer to invoke the desired protocols. These messages encapsulate information about the protocol to be invoked and are termed Protocol Triggers or Triggers (see Section 4.3).

Messages are sent from the `service.control-tower` via publish-subscribe messaging, an asynchronous communication paradigm where publishers and subscribers are loosely coupled [16]. Redis, a fast and open-source key-value data store, is used as the message broker to enable this. Given the real-time, event-driven and one-to-many nature, publish-subscribe messaging is a highly effective choice for efficient and scalable communication between the Control Plane and Operator Layer.

To watch for events, one needs to develop a plugin for the `service.control-tower` that integrates with software in the Management Layer. In this project, a plugin for Riffyn Nexus was developed which periodically polls its API to identify when *Experiment Runs* have been started and uses the input data from those *Runs* to generate the appropriate Protocol Triggers to be issued.

Data Plane: The Data Plane is responsible for forwarding data from the Operator Layer back up to the Management Layer. Services in the former can export data to the `service.data-gateway` via HTTP POST requests, which will then forward them to upstream destinations via its plugins. Similar to the Control Plane, one needs to develop a plugin for the `service.data-gateway` that integrates with the Management Layer. In this project, a plugin for Riffyn Nexus was developed that knows how to relay data back to their associated *Runs* by tracing universally unique identifiers (UUIDs) issued by the `service.control-tower`. This closes the feedback loop between inputs and outputs throughout the experiment.

4.2.3 Operator Layer

The Operator Layer consists of the services that interface with automation targets in the Device Layer and are also known as Operators. Each Operator essentially provides a form of wrapper for interacting with its target device and implements a set of Protocols that can be invoked for that device. These Operators are then responsible for receiving Triggers from the Control Plane and

responding accordingly for Protocols they implement. Operators are also responsible for exporting data from the underlying device to the Management Layer via the Data Plane if required.

Depending on the underlying device, the means of integration will vary between Operators - some might function through APIs, others through the file system, system calls, or even serial communication. It is thus common for Operators to be deployed on edge hosts.

Additionally, the Operator Layer also contains auxiliary services that are internally depended upon by other services. In the current architecture, Redis is being used as a distributed cache. This serves as a lightweight data store and also aids failure recovery - if a service fails midway through a process and is rebooted, rather than lose all its progress, it will be able to recover data from checkpoints as long as they have been cached [17].

4.2.4 Device Layer

The Device Layer consists of the devices to be integrated into the ULAS platform via their Operators. These are the "workers" that carry out the actual execution of experimental protocols, such as liquid handling robots, plate readers, microcontrollers, etc. They thus form the modular building blocks that enable one to compose more complex automation workflows.

Contrary to its name, "device" in this context doesn't strictly refer to physical hardware. The term is used to encompass software, firmware, as well as hardware. In fact, most devices have some of each aspect (ie. software that controls hardware through firmware).

4.3 Protocol Definition Language

To invoke experimental protocols on automation targets in a standardised and device-agnostic manner, a common communication protocol needed to be established. Thus, the Protocol Definition Language (PDL) was developed, forming the basis for communication between the Control Plane and Operators. At its core, the PDL describes a standardised message format that Operators expect from the Control Plane. It is based on JavaScript Object Notation (JSON), which was chosen as it is a platform-agnostic, human-readable and widely used data format [18]. Using JSON also means that one can easily mock these messages during the development and testing of Operators. The PDL is made up of Protocols and Protocol Triggers, illustrated in Figure 5.

```
{
    "apiVersion": "ChiBio/v1alpha1",
    "protocol": "Bioreactor",
    "spec": {
        "devicePosition": "M0",
        "deviceName": "Hydrogen",
        "od": 0.42,
        "volume": 50,
        "thermostat": 38.8,
        "fp1Excite": "6500K",
        "fp1Gain": "8x"
    }
}
```

Figure 5: Trigger for the `Bioreactor` Protocol Implemented by the ChiBio Operator

4.3.1 Protocols

Protocols can be thought of as predefined "functions" that an Operator is able to perform, comparable to blueprints for experiments. These form the API surface of Operators and are implemented through Operator-specific code responsible for interacting with the underlying device. For instance, the Operator for a plate reader might implement a Protocol named `MeasureAbsorbance` that runs an absorbance measurement on the plate reader when invoked.

Similar to functions in code, each invocation of a Protocol should be parameterisable to tune its execution. Thus, every Protocol needs to define a Protocol Definition, analogous to a function signature. Each definition is a JSON schema documentation that includes the Protocol name as well as a `spec` definition, which defines the list of arguments that the Protocol accepts - these are the keys one can specify in the `spec` object of a Trigger, essentially forming the API contract for invoking that Protocol. Using the example above, the `MeasureAbsorbance` Protocol might have a `spec` definition containing a `measurementWavelength` property for configuring the wavelength of light to be used.

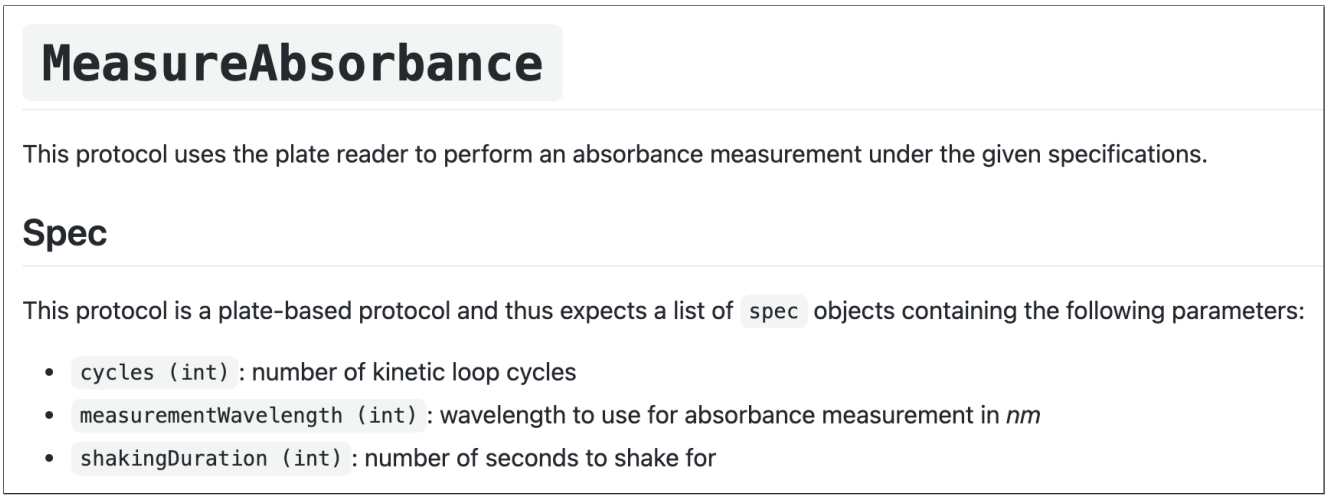

Figure 6: Protocol Definition for the `MeasureAbsorbance` Protocol

4.3.2 Protocol Triggers

Triggers are the JSON messages sent by the Control Plane to their intended Operators as a declarative way of triggering a desired Protocol. Each Trigger needs to contain the `apiVersion` and `protocol` to be invoked, where the former is used as the publish-subscribe topic. It should also contain a `spec` object containing the key-value pairs specific to that Protocol. Operators will act accordingly on each Trigger it receives, using the values defined in their `spec`. Continuing from the above example, a Trigger for `MeasureAbsorbance` could contain a `spec` object containing the key-value pair `measurementWavelength`: `600` to execute the absorbance measurement using light of wavelength 600 nm.

4.3.3 Plate-Based Protocols

Some protocols might involve a plate with wells, common in synthetic biology experiments. To support this, plate-based Protocols by convention should accept a JSON list of nested `spec` objects instead, where each object is associated to the well at its corresponding index, starting from 0.

The convention for indexing wells on a plate increases column-first, always starting from the top-left corner and ending in the bottom-right corner (Figure 7).

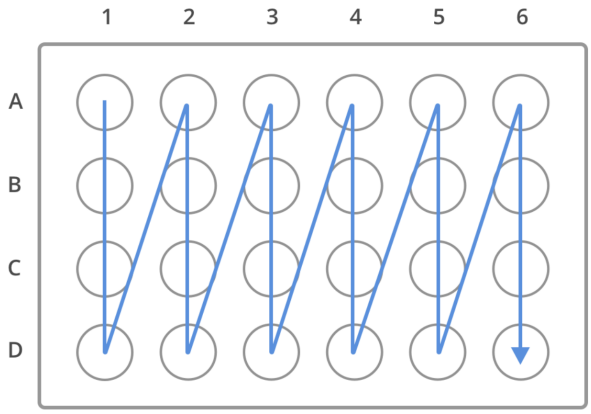

Figure 7: Well Indexing Conventions for Plate-Based Protocols [2]

This means that for a 96-well plate (8x12), well A1 will correspond to the `spec` object at index 0, B1 to 1, and so on until 95. This is demonstrated in Figure 8.

```
{
    "apiVersion": "OT-2/v1alpha1",
    "protocol": "SimpleDilution",
    "spec": [
        { "volume": 150 },
        { "volume": 150 },
        { "volume": 100 },
        { "volume": 100 }
    ]
}
```

Figure 8: Trigger for Operating on 4 Wells (A1 to D1)

4.3.4 API Versioning

To ensure backwards compatibility and ease of upgrading Operators, an API versioning system was introduced - a common practice for developing robust APIs [19]. As illustrated by the `apiVersion` in Figure 8, this versioning syntax contains the name of the Operator (`OT-2`) followed by a version (`v1alpha1`). Given that a Protocol's contract might change over time, one needs to specify the exact API version when invoking a Protocol.

This versioning system can be further extended to support multiple devices of the same API. By convention, this can be achieved by suffixing API versions with a unique name for each device unit and running multiple instances of an Operator tied to each of those devices. For instance, to support two Tecan Spark® plate readers, they can be namespaced as `TecanSpark/v1alpha1/Spark1` and `TecanSpark/v1alpha1/Spark2` respectively. This way, to invoke a Protocol on a specific device, one can provide the fully qualified API version suffixed with the device's unique name.

4.3.5 Implemented Protocols

This standardised Protocol system provides a set of common interfaces and conventions for easily integrating new devices into ULAS - one just needs to develop an Operator for the intended device and implement the Protocols to be exposed, including documentation of their respective

Protocol Definitions. Protocols can then be invoked by issuing the appropriate Triggers from the Routing Layer. All the Protocols implemented in this project for the chosen automation targets are detailed in Table 2:

Operator	**API Version**	**Protocol**	**Spec**
ChiBio	`ChiBio/v1alpha1`	`Bioreactor`	`deviceName, devicePosition, od, volume, thermostat, fp1Excite, fp1Gain`
TecanSpark	`TecanSpark/v1alpha1`	`MeasureAbsorbance`	[`cycles, measurementWavelength, shakingDuration`]
OT-2	`OT-2/v1alpha1`	`PlateTransfer`	[`volume`]
		`SimpleDilution`	[`volume`]

Table 2: Protocol Definitions of Implemented Protocols ([Plate-Based])

4.4 Operator Implementations

This section provides a high-level overview of the Operators that were developed for the chosen automation targets, in terms of how they were interfaced with the underlying devices.

4.4.1 ChiBio

The `service.chibio-relay` is the Operator for the Chi.Bio, written in Python. This Operator currently provides a `Bioreactor` Protocol and uses the values provided in the `spec` of received Triggers to configure an experiment to be ran on the Chi.Bio device. It does this via API endpoints on the Chi.Bio server - comparable to how users would normally input values via its GUI (Figure 9). The Operator doesn't actually start experiments automatically, instead requiring users to still manually click "Start" on the GUI - it only sets the desired configuration in anticipation. This was deliberate as it might pose a safety hazard for the Chi.Bio to be started automatically without any human supervision.

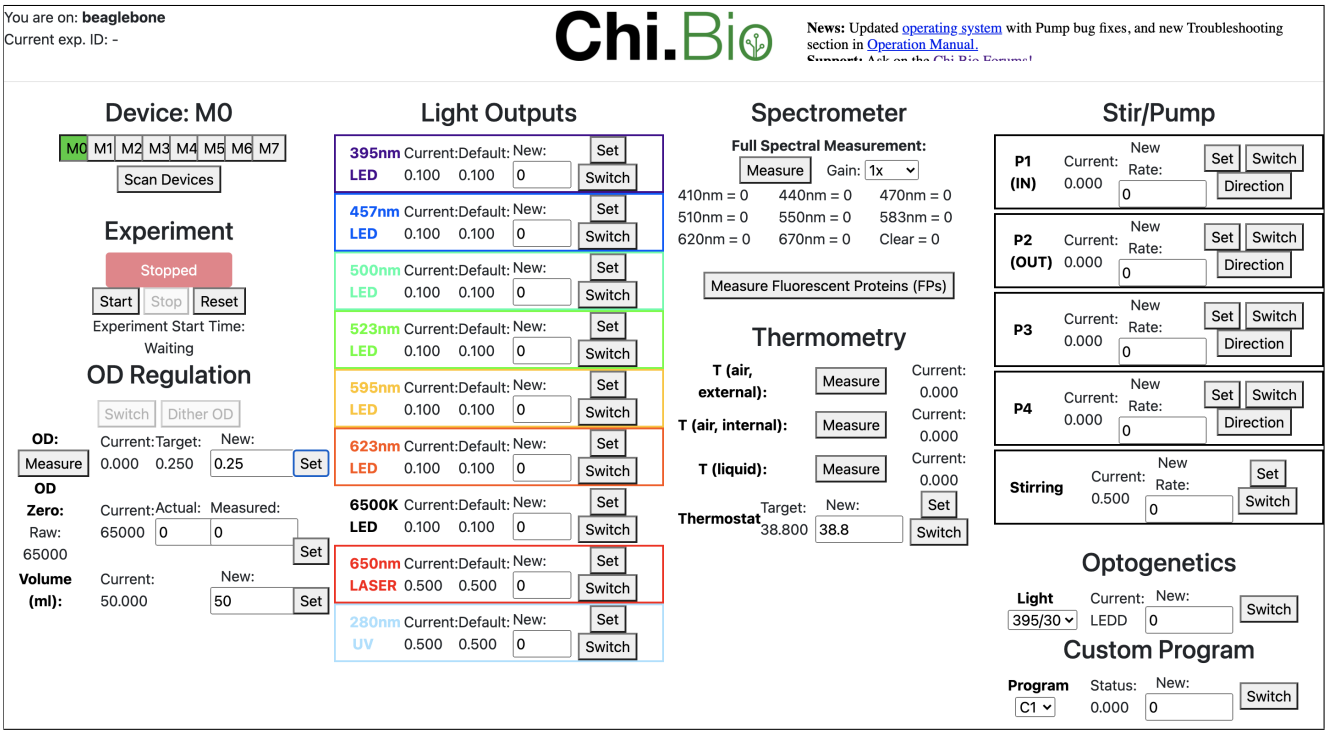

Figure 9: Chi.Bio GUI Populated via Triggers

Separately, the Operator was also programmed to periodically parse CSV data files generated by the Chi.Bio system, forwarding any new data to the Data Plane. This means that data can be streamed from the Chi.Bio to the Routing Layer in near real-time.

The Operator currently only works with a fork of the Chi.Bio software that exposes the above API endpoints, available at `https://github.com/jace-ys/ChiBio/tree/develop`.

4.4.2 TecanSpark

The `service.tecan-spark-relay` is the Operator for the Tecan Spark®. It is written in C#, the language of the SparkControl Automation Interface API used for interacting with the plate reader programmatically. Each Protocol implemented for this Operator accepts a set of values to be provided in the `spec` of associated Triggers, which are then used to populate its corresponding Spark Method XML template (Figure 10). This means that each received Trigger is used to generate a complete Spark Method XML containing all the instructions and configuration needed by the plate reader to perform a measurement, which is then passed to the API to be executed on the device. Once the measurement has completed, the Operator retrieves the XML data file generated by the plate reader via the API and parses it, thereafter forwarding the measurement data to the Data Plane.

```
<AbsorbanceStrip MeasurementsCount="1" WavelengthMeasurement="{{ spec.measurementWavelength }}0">
    <AbsorbanceStrip.DataLabels>
        <DataLabel Type="Measurement" Unit="OpticalDensity" />
    </AbsorbanceStrip.DataLabels>
</AbsorbanceStrip>
```

Figure 10: Snippet of Spark Method XML Template for `MeasureAbsorbance` Protocol

4.4.3 OT-2

The `service.ot-builder` is the Operator for the Opentrons OT-2 written in Python. Each Protocol implemented for this Operator accepts a set of values to be provided in the `spec` of associated Triggers, which are then used to populate its corresponding OT-2 Python protocol template (Figure 11). The complete protocol file can then be simulated and downloaded, before uploading to the Opentrons App for execution on the liquid handling robot.

```python
def run(protocol: protocol_api.ProtocolContext):
    tiprack = protocol.load_labware(**config["tiprack"])
    pipette = protocol.load_instrument(**config["pipette"], tip_racks=[tiprack])

    source = protocol.load_labware(**config["source"])
    destination = protocol.load_labware(**config["destination"])

    for (index, well) in enumerate(spec):
        volume = well["volume"]
        pipette.transfer(
            volume=volume,
            source=source.wells()[index],
            dest=destination.wells()[index],
            new_tip="always",
            mix_after=(3, volume / 2),
        )
```

Figure 11: OT-2 Python Protocol Template for `PlateTransfer` Protocol

Additionally, the Operator also exposes a web frontend to allow users to specify the OT-2 deck configuration to use for each Trigger (Figure 12), which are also used in populating templates. This is necessary as templates may need to know the physical configuration of the deck decided by the user, a separate set of parameters from the values provided in the Trigger. This draws a distinction between operational configuration (the former) and experimental parameters (the latter), a deliberate choice that was made to decouple Protocols from device-specific configuration.

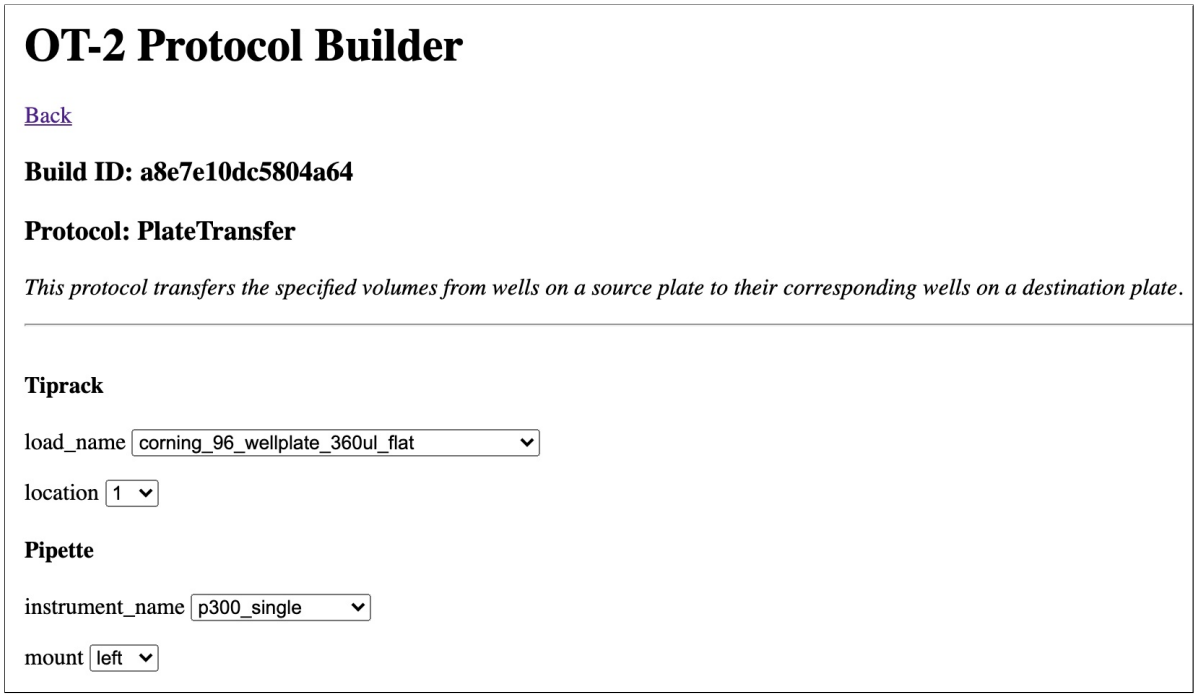

Figure 12: Specifying OT-2 Deck Configuration via the OT-2 Operator

4.5 Development and Testing

Mocking is a common practice used in software development to aid the testing of integration code. Hence, to enable rapid prototyping of Operators without the need for directly connecting to physical devices in the lab, which could be cumbersome and inconvenient, mock implementations that imitate the behaviour of the actual software to be integrated with were used. For instance, a mock Chi.Bio server was developed to facilitate the development of the Chi.Bio Operator, allowing one to interact with the Chi.Bio GUI in the same way as the actual software but without any hardware side effects. Additionally, to be able to conveniently test the behaviour of Operators in isolation from the rest of the ULAS infrastructure, a small Python script was created for simulating Triggers by directly sending JSON messages to the publish-subscribe message broker, essentially mocking what the Control Plane does.

4.6 Deployment

To make the installation and deployment of ULAS as straightforward and repeatable as possible, each service was containerised using Docker. Containers provide "application isolation, cost-effective scalability, and disposability" while Docker has increasingly become the industry standard for containerisation as it enables developers to easily package, ship, and run applications in a consistent and portable manner [20].

The Core services of ULAS were deployed to a server within the lab and managed using Docker Compose, a container orchestration tool. The Edge services were deployed on the respective machines that they are interfaced with - the ChiBio and TecanSpark Operators were deployed to the hosts that are connected to their intended devices. To enable Edge services to be able to communicate with Core services over the network, several firewall ports on the server needed to be configured to accept incoming and outgoing Transmission Control Protocol (TCP) traffic.

4.7 Open-Source

In line with the goal of open-sourcing the project, all the code and documentation for ULAS is available on GitHub at `https://github.com/jace-ys/lab-automation`. The codebase was also designed and organised in a way that facilitates adding new plugins and integrations.

5 Results

5.1 Application to Synthetic Biology

Using the Protocols provided by the Operators for the devices integrated into ULAS as building blocks, one can start to develop automation around common experimental workflows used in synthetic biology. To test ULAS end-to-end from the perspective of a researcher using the platform, the following workflows were designed and encoded in Riffyn Nexus, based on regular use cases of researchers within the lab:

- Culturing cells in a bioreactor (Figure 13)
- Performing an optical density (OD) normalisation on a cell culture (Figure 14 and 15)

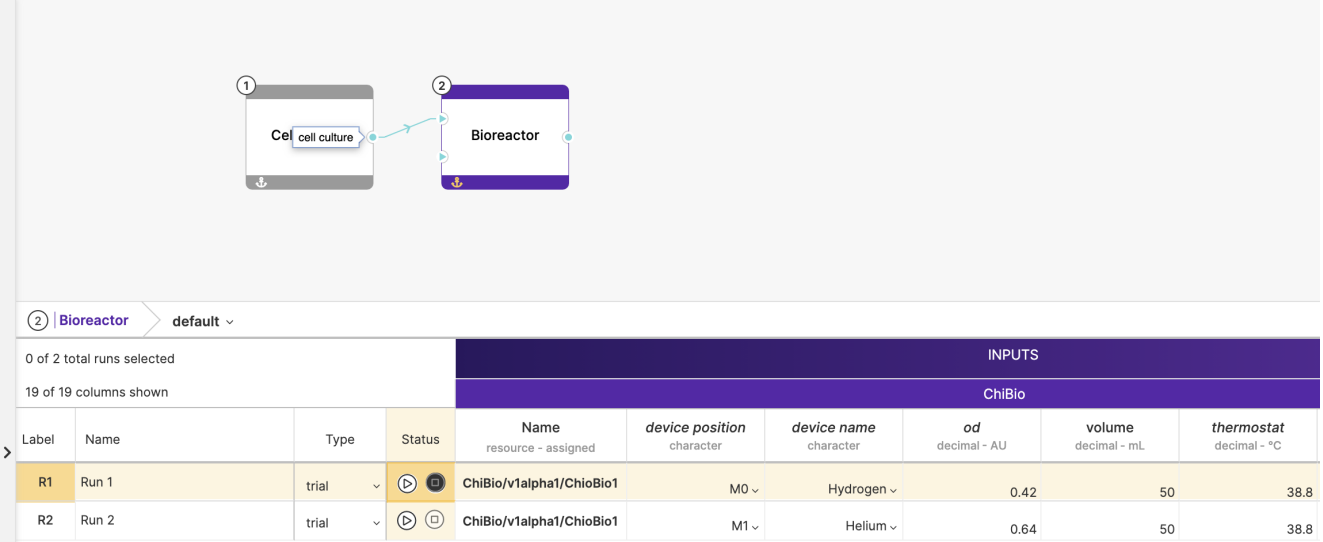

Figure 13: Culturing Cells in Bioreactor - Bioreactor Step Automated via the ChiBio Operator's `Bioreactor` Protocol

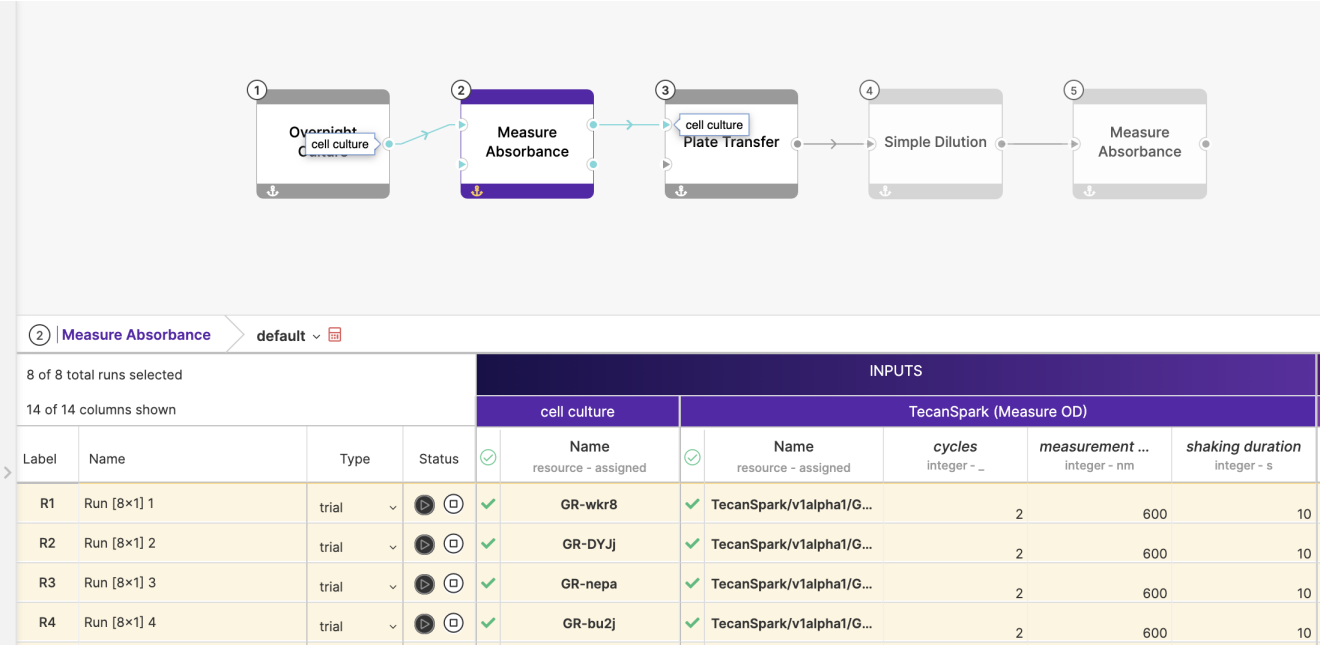

Figure 14: OD Normalisation - Measure Absorbance Step Automated via the TecanSpark Operator's `MeasureAbsorbance` Protocol

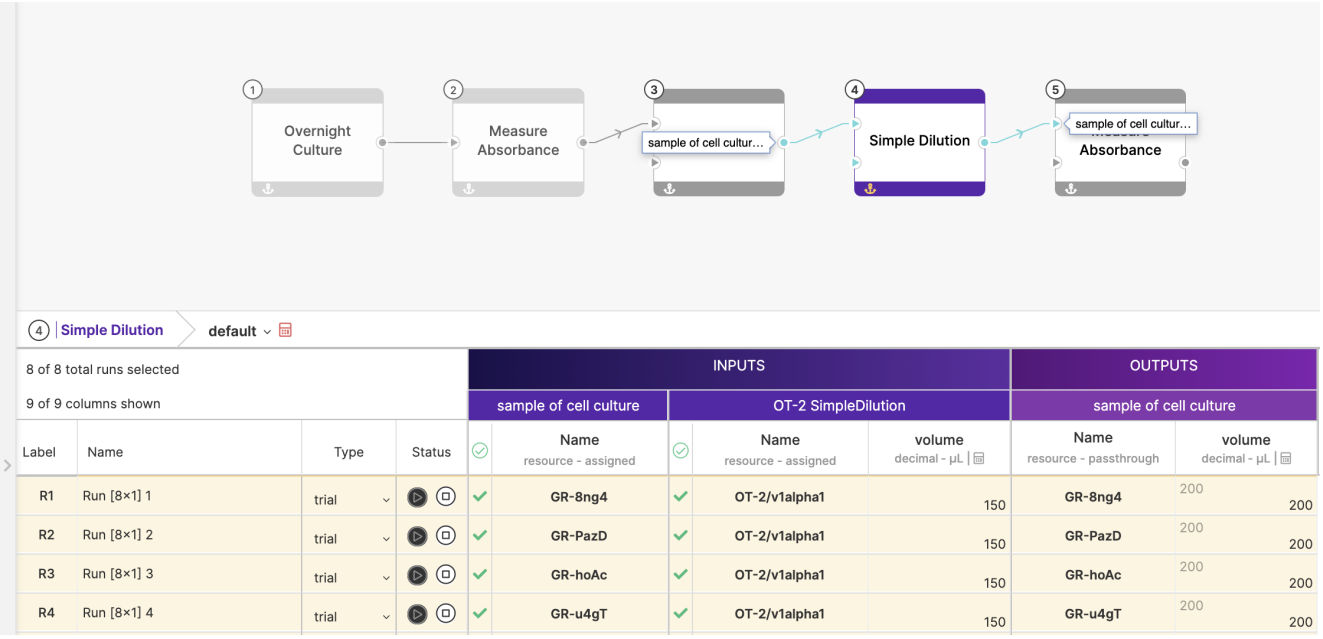

Figure 15: OD Normalisation - Simple Dilution Step Automated via the OT-2 Operator's `SimpleDilution` Protocol

Using the experiments designed above, when the "start" button for a *Run* was clicked (or multiple runs in the case of plate-based Protocols), ULAS successfully triggered the respective Protocol on the desired automation target, additionally passing it the input data provided for the *Run(s)*. The output data produced by the devices were then automatically collected and successfully written back into the output of their associated *Run(s)*. These data could then be propagated or used to calculate derived data for downstream steps (shown in Figure 15), providing seamless and closed-loop automation workflows that spanned multiple devices.

A video demonstrating an automated OD normalisation workflow can be viewed at `https://www.youtube.com/watch?v=jMFYsWYvMZE`. Throughout the automation process, minimal user intervention was required except where necessary. For instance, in the Measure Absorbance step, the user was required to place the plate to be measured into the plate reader before clicking the "start" button; users only need to be concerned with the physical setup of devices and providing intermediate input if required, leaving the movement of data entirely to ULAS.

Compared to existing means of performing an OD normalisation manually, the workflow automated by ULAS offers the following key benefits for researchers:

- Riffyn Nexus forms the single point of contact for defining experimental parameters and recording associated data, providing end-to-end provenance over experimental data. Being cloud-based also means one can access data from outside the lab.
- No need for manual extraction of data from spreadsheets as data collection is automated. Additionally, one does not need to manually calculate derived data, which can be done entirely through Riffyn Nexus.
- Experiment protocols can be kicked-off directly through Riffyn Nexus, most of which will be hands-free besides having to provide some supervision and place equipment in the right locations. This reduces the likelihood of errors from having to manually configure devices or carry out experiments by hand.
- Greater standardisation throughout the lab in how experiments are carried out and data is collected as researchers utilise the same best practices and platform for conducting their experiments.

6 Discussion

6.1 Outcomes

Through the development of ULAS, three significantly different devices have been successfully integrated into a common platform for lab automation with standardised interfaces and communication protocols, additionally powered by Riffyn Nexus that provides a high-level and comprehensive GUI for managing the lifecycles of automated experiments. Application to synthetic biology has demonstrated the potential for ULAS to help researchers achieve greater reproducibility and scalability in their work, which is likely to grow exponentially as more integrations get added and the platform matures.

One of the primary advantages ULAS has over existing solutions is that it doesn't introduce new tools that researchers have to learn and familiarise with; researchers can continue to use the ones that they already use as ULAS just creates a more integrated and cohesive environment between devices and data in the lab. ULAS was also built to be completely customisable, meaning that labs can plug-and-play based on their own specific needs or use it as a framework to build upon. This is highly advantageous given that no two labs have the same equipment and routines.

Additionally, ULAS actively employs software engineering best practices and system design thinking that are prevalent in the IT industry for building modular and scalable software systems. Examples include microservices architectures, containerisation, API versioning, structured logging, continuous integration, and the Twelve-Factor App methodology [21]. These not only make ULAS more robust but also serve as good guidelines for open-source contributors.

6.2 Limitations and Future Work

One current limitation is that ULAS is only as useful as the number of integrations available; a more battle-tested ecosystem is required before wider recognition and adoption can be achieved, which requires more time and upfront investment into the project in the short term. Additionally, given that the project was largely developed with a few specific integration targets in mind, it might not be able to sufficiently cover every use case at its current stage. For instance, there might be better alternatives to Riffyn Nexus that are open-source and less opinionated. However, as more integrations get added and more use cases arise, future iterations of the project can be gradually fine-tuned to achieve greater standardisation and generalisability.

Additionally, the accuracy and precision of experiments performed by ULAS is still largely dependent on the underlying devices being used and is not something directly addressable by the platform. However, what it provides instead is the flexibility to easily switch between integrations. For instance, one might want to replace Opentrons with a Hamilton liquid handler for greater accuracy and precision. This can be done without having to significantly change the way experimental protocols are encoded, as long as an integration exists for it with similar interfaces.

Unsurprisingly, ULAS has not yet been load tested for industrial applications. The current architecture might not be able to tolerate the required workload for such use cases; using Redis as a publish-subscribe message broker could likely be a bottleneck as Redis is meant to be a lightweight solution. To resolve this, one could use more fault-tolerant and high-throughput message queues in place of Redis, such as Apache Kafka. However, Kafka was not chosen for this project as it requires specialised knowledge to operate and maintain.

Finally, creating new integrations and Protocols currently requires some level of software development knowledge, given the complex architecture behind the platform. This might create a barrier to entry for researchers who lack the technical expertise and know-how to extend ULAS. This can hopefully be addressed in the future through providing internal tools and libraries that contributors can use to easily generate code or bootstrap new features.

7 Conclusion

This paper has addressed the motivations and implementation behind ULAS, a universal lab automation system built to be modular, scalable, extensible, open-source, and cost-effective from the ground up. Through standardised interfaces and communication protocols, it allows one to unify the devices and data within one's lab environment into a centralised software platform, which form the foundation for one to build seamless, data-centric, and closed-loop automation workflows. This paper has also demonstrated how ULAS was harnessed to automate some common experimental protocols used in synthetic biology, displaying promising results that only scratch the surface of what's possible with such a platform. Given that the success of ULAS hinges greatly on a thriving ecosystem and wider recognition, the hope is for the broader community to build upon the project or even inspire new approaches to foster greater interoperability and cohesion between the vast and disparate ecosystem of instruments and tools used in research, with the ultimate goal of helping researchers achieve greater reproducibility and scalability through automation whilst also improving their lives so that they can focus on what they do best: make sense of the science.

8 References

[1] M. M. Jessop-Fabre and N. Sonnenschein, "Improving reproducibility in synthetic biology," *Frontiers in Bioengineering and Biotechnology*, vol. 7, 2019.

[2] Opentrons Labworks Inc., "Labware - Opentrons Python API V2 Documentation." `https://docs.opentrons.com/v2/new_labware.html#well-ordering`. Accessed: 12 Apr 2021.

[3] M. Baker, "1,500 scientists lift the lid on reproducibility," *Nature*, vol. 533, 2016.

[4] L. P. Freedman, I. M. Cockburn, and T. S. Simcoer, "The economics of reproducibility in preclinical research," *PLoS Biology*, vol. 13, 2015.

[5] T. Fell, S. Ward, M. Gershater, M. Watson, P. Crane, and R. Wiederhold, "Computer-Aided Biology," white paper, Synthace, 10 2018.

[6] P. Zucchelli, G. Horak, and N. Skinner, "Highly Versatile Cloud-Based Automation Solution for the Remote Design and Execution of Experiment Protocols during the COVID-19 Pandemic," *SLAS Technology*, vol. 26, pp. 127–139, 4 2021.

[7] A. Wolf, P. Galambos, and K. Szell, "Device Integration Concepts in Laboratory Automation," 2020.

[8] H. Bär, R. Hochstrasser, and B. Papenfuß, "SiLA: Basic standards for rapid integration in laboratory automation," *Journal of Laboratory Automation*, vol. 17, 2012.

[9] T. Decoene, B. D. Paepe, J. Maertens, P. Coussement, G. Peters, S. L. D. Maeseneire, and M. D. Mey, "Standardization in synthetic biology: an engineering discipline coming of age," 2018.

[10] J. L. Johnson, H. tom Wörden, and K. van Wijk, "PLACE: An Open-Source Python Package for Laboratory Automation, Control, and Experimentation," *Journal of Laboratory Automation*, vol. 20, 2015.

[11] N. F. Delaney, J. I. Echenique, and C. J. Marx, "Clarity: An open-source manager for laboratory automation," *Journal of Laboratory Automation*, vol. 18, 2013.

[12] I. Schmid and J. Aschoff, "A scalable software framework for data integration in bioprocess development," *Engineering in Life Sciences*, vol. 17, 2017.

[13] B. Miles and P. L. Lee, "Achieving Reproducibility and Closed-Loop Automation in Biological Experimentation with an IoT-Enabled Lab of the Future," *SLAS Technology*, vol. 23, pp. 432–439, 10 2018.

[14] M. Crone, P. Randell, Z. Herm, S. Missaghian-Cully, L. Perelman, P. Pantelidis, and P. Freemont, "Design and Implementation of An Adaptive Pooling Workflow for SARS-CoV-2 Testing in an NHS Diagnostic Laboratory," *SSRN Electronic Journal*, 2021.

[15] M. Fowler, "Microservices." `https://martinfowler.com/articles/microservices.html`, 2014. Accessed: 20 Dec 2020.

[16] M. O'Riordan, "Publish-Subscribe: Introduction to Scalable Messaging." `https://thenewstack.io/publish-subscribe-introduction-to-scalable-messaging`, 2014. Accessed: 20 Dec 2020.

[17] J. DeVale, "Checkpoint-Recovery." `https://users.ece.cmu.edu/~koopman/des_s99/checkpoint/`, 1999. Accessed: 14 Dec 2020.

[18] J. Freeman, "What is JSON? A better format for data exchange." https://www.infoworld.com/article/3222851/what-is-json-a-better-format-for-data-exchange.html, 2019. Accessed: 11 Jan 2021.

[19] J. Simpson, "Everything You Need to Know About API Versioning." https://www.ibm.com/cloud/learn/docker, 2021. Accessed: 16 Feb 2021.

[20] IBM Cloud Education, "What is Docker?." https://www.ibm.com/cloud/learn/docker, 2020. Accessed: 3 Mar 2021.

[21] A. Wiggins, "The Twelve-Factor App." https://12factor.net/, 2011. Accessed: 8 Jan 2021.

9 Acronyms

API application programming interface. 1, 5, 8–15, 20

GUI graphical user interface. 2, 7, 10, 14, 15, 17, 20, 25

HTTP Hypertext Transfer Protocol. 9, 10

IoT Internet of Things. 4, 5

JSON JavaScript Object Notation. 11, 12, 17

LIMS laboratory information management system. 5, 25

OD optical density. 2, 18, 19

PDL Protocol Definition Language. 11

TCP Transmission Control Protocol. 17

UUID universally unique identifier. 10

A Evaluation of Existing Solutions

Solution	Advantages	Disadvantages
Strateos (previously Transcriptic)	• Offers a cloud lab service that grants remote access to the capabilities of an automated laboratory. • Developed Autoprotocol, an intuitive language specification for specifying experimental protocols that is platform-agnostic.	• Cloud labs are only able to support approximately 3% of end-to-end workflows required by published papers within biomedical research [1]. • Autoprotocol is only a data specification and thus lacks direct means of implementing automation.
Synthace	• Their proprietary cloud platform, Antha, provides device drivers for various automated liquid handling platforms. • The Antha GUI allows one to plan and execute multi-factorial experiments with automated data aggregation.	• Is largely limited to liquid handling systems at its current stage. • Is closed-source and is thus not extensible with custom device integrations.
Aquarium	• Provides the Aquarium LIMS for representing experimental protocols as executable code and designing workflows that yield clear instructions for lab technicians to follow.	• Has a complex and unintuitive user interface that takes time to familiarise with. • Not scalable as it still relies on a human user to perform the actual execution of experimental protocols.
SiLA	• Established by a consortium of industry players. • The proposed standards provide clear directives on developing SiLA-compliant devices.	• Relies on hardware interfaces that can be difficult to scale and debug. • Is more focused on integration and thus lacks direct means of implementing automation.

Table 3: Evaluation of Existing Solutions