A COLLABORATIVE SOFTWARE TOOLCHAIN FOR AUTOMATIC COLLECTION AND COMPARATIVE ANALYSIS OF SENSOR CHARACTERIZATION DATA

By

Charles Samuel Boling

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Electrical Engineering – Master of Science

2016

ABSTRACT

A COLLABORATIVE SOFTWARE TOOLCHAIN FOR AUTOMATIC COLLECTION AND COMPARATIVE ANALYSIS OF SENSOR CHARACTERIZATION DATA

By

Charles Samuel Boling

Reproducible research has been recognized as a growing concern in most areas of science. To achieve widespread adoption of repeatable, transparent research practices, some commentators have identified a need for better software for authoring reproducible digital publications. Complicating this goal, scientific investigations increasingly involve interdisciplinary teams, sophisticated workflows for acquiring and analyzing data, and huge datasets that rely on considerable metadata to interpret. Computational scientists have begun to adopt tools for managing the complex histories of their data and procedures, but software which simultaneously allows researchers to specify experiments, remotely control equipment, and capture and organize data remains immature. This thesis demonstrates a software architecture for programmable remote control of custom and commercial lab equipment, automatic annotation and queryable storage of data sets, and provenance-aware specification of experiment and analysis procedures. The design consists of a suite of small, single-purpose software services which may be controlled remotely from a web browser, including a graphical programming tool, an abstraction layer for interfacing with commercial and custom embedded systems, and a hybrid document/table database for persistent storage of annotated experimental data. The software implementation embraces modern web technologies and best practices to produce a modular, user-extensible framework that is well-suited for helping to integrate computer-controlled research labs with the emerging Internet of Things.

ACKNOWLEDGMENTS

I am especially thankful for the tremendous support of my family and loved ones throughout my time at school. In particular, I want to express my appreciation for Paula, whose patience, feedback, and reassurance have been invaluable in much more than just my academic work.

The software described in this thesis would not have been possible to realize without the benefit of many prolonged design discussions with student researchers Ian Bacus and Yousef Gtat, and I am tremendously grateful for their input and development support. Furthermore, the entire AMSaC lab has been a great help and a fine community to work in the last few years.

This work was partially supported by funding from NIH grant 1R01ES022302.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

MOTIVATION

Poor reproducibility of scientific publications has been recognized as a growing problem in a number of research areas, particularly those in which experiments are complex and sensitive to small variations in methodology. Large-scale replication efforts have suggested reproducibility rates in some fields as low as 10% [8]. Researchers have begun to call for improved documentation of the scientific process in fields as diverse as experimental psychology [17], pharmaceutical research [8], astronomy [1], and areas of computer science such as machine learning [11]. Much of the recent attention paid to this issue comes in response to unique characteristics the scientific process has developed in the digital age, including unprecedented publication volume, insufficient documentation of complex experimental procedures, and the availability of software packages for manipulating statistics, but some critics suspect that many long-standing results are also inadequately verifiable. It has been suggested that factors such as publishing pressure, confirmation bias, and inadequate statistical power of hypotheses promote the widespread publication and citation of unverifiable claims in all areas of experimental science [36]. To make matters worse, the last few years have seen many cases of peer review fraud [25], outright data falsification [24], and ethically dubious activities such as p-hacking [34], the practice of massaging data to cross the accepted threshold of statistical significance.

In addition to these systemic problems with the publication process, the day-to-day practice of modern science and engineering research presents a steadily growing host of data organization challenges to investigators. Often an experimental program involves many personnel, each with a unique specialization and research focus, performing interdependent experiments at multiple universities. Each such experiment is the product of a huge host of influences, and data are often collected in ad-hoc or incompatible formats, making it difficult to draw honest comparisons between results or to isolate methodological problems. Especially in the case of technology development and exploratory research, it is desirable for researchers in these kinds of large projects to use uniform data acquisition protocols, unambiguously describe their experimental procedures, and collate their work into self-contained, consistently formatted units for distribution to collaborators.

One proposition for improving the reproducibility of future scientific work is to better standardize documentation practices for laboratory procedures [36] and ultimately to transition from traditional paper-based publication models to electronic formats which capture the intricacies of modern scientific work. Ideally, a unit of disseminated research would provide enough detail for future researchers to replicate every step of the experiment and analysis associated with a publication and for reviewers to identify sources of errors, details warranting further examination, and academic misconduct. Confirmation studies as well as exploratory research could benefit from the adoption of flexible software tools for collecting data and chronicling experimental procedures. Particularly in "*in-silico*" fields, where experiments consist of the transformation and analysis of data sets within the digital domain, research stands to benefit from software that can automate and standardize tasks such as experimental design and record keeping, and some publication organizations have begun to encourage sharing of code, procedures, and raw data alongside submitted manuscripts. Software tools for managing complex simulation and data analysis pipelines have begun to emerge in recent years which offer support for a number of powerful features, including data provenance, sharing and refining workflows, and packaging execution environments into virtual machines for later execution on different hardware [28, 53]. These tools typically do not attempt to model or automate non-software research tasks in detail. To address some of the informatics challenges of more "hands-on" research, several companies have developed so-called laboratory information management system (LIMS), which are better suited to the inventory and data management needs of traditional scientific facilities such as wet labs. However, many scientific endeavors involve some mixture of structuring *in-silico* analysis workflows and directly manipulating physical systems, and software toolchains for uniformly managing procedures of this nature remain immature.

In fields where research involves both sophisticated software analysis and intensive batteries of physical experiments, investigators could benefit from a software platform which unifies protocol design, data acquisition, result annotation and archiving, signal processing, and other tasks involved in the complete research and development life cycle. Such a tool should be (i) automatic, employing computer control whenever possible to produce organized, uniform and repeatable experiments; (ii) extensible and modular, promoting adoption of new equipment, experimental methods, and data analysis techniques via user-crafted plugins; (iii) collaborative, allowing results and proposed experiments to be shared, annotated, and reviewed at many levels of detail; (iv) bespoke, accommodating and complementing the focus of each researcher involved in an interdisciplinary research and development project, and (v) provenance-aware, enabling fine-grained differential analysis of experimental outcomes and methodologies. Existing approaches do not combine data acquisition and archiving features with interfaces for process customization in a way that meets all the above goals. In many cases tools built for these purposes are also insufficiently adaptable for the fast-paced and varied needs of active scientists, causing users to abandon the software once it presents more limitations than benefits.

Fortunately, modern web technologies have begun to enable software design strategies that make a complex, customizable end-to-end solution feasible. Network-enabled services with diverse purposes and internal infrastructures have become increasingly interoperable thanks to the adoption of self-documenting web application programming interface (API)s. The increasing sophistication of web browsers has allowed for an explosion of rich client-side software experiences, enabling full-featured and user interfaces which are platform-independent and easily updated. A number of technologies such as distributed version control, demand-scaling cloud hosting services, and real-time full-duplex network data streaming have emerged as powerful tools for rapidly building robust and flexible web applications with unprecedented capabilities. The availability of inexpensive sensor and network hardware has begun to spur the growth of the emerging Internet of Things (IoT), a vision of the near future in which ubiquitous computing devices collect data, communicate with each other, and interact with their environments. Together these advancements provide a rich software ecosystem for implementing a next-generation LIMS for performing complex experiments, curating detailed data sets, and generating publication units with end-to-end reproducibility.

This thesis describes the design and implementation of a suite of software tools for data acquisition and provenance tracking with the goal of leveraging computer automation to create a scientific dissemination format with rich facilities for comparing results, identifying new directions of research, and fostering collaboration than traditional print publications. The design of the described software tool embraces modern web technologies, separating functional units into independent networked services which communicate by discoverable web APIs. This architecture enables investigators to interact with each other's research remotely and to independently create reusable services of their own. The core components of the system are modular and loosely coupled, and users are encouraged to modify, create, and share software components to meet the unique needs of their research. By designing a modular architecture which anticipates rapidly changing requirements and enables users to take an active role in software maintenance, the platform is intended to grow with its user base and enjoy broader usefulness and greater longevity than existing free and commercial lab informatics packages. The framework provides high-level capabilities for remotely controlling lab equipment and routing captured sensor data, with a vision of connecting research labs to the nascent IoT. To demonstrate and explore the system's capabilities, a embedded system was developed for performing customizable electrochemical experiments, which includes a multichannel arbitrary waveform generator. The system's architecture is described in detail along with an overview of the developers' implementation choices. Throughout the exposition, the design and integration of multiple custom electrochemical instruments serve to demonstrate how users might add and modify software components to meet the needs of their own research.

The following chapters explore the existing space of software tools, lay out the project's design goals describe the high-level structure of the design, and explain the techniques and technologies used to implement the completed system. Finally we describe the characteristics of the design that we feel are unique or notable. Due to the large volume of technical terminology involved in discussing scientific software tools and software technologies, a glossary of terms is included at the end of the thesis.

CHAPTER 2

BACKGROUND

A growing body of work in the field of meta-research has identified a number of obstructions to research reproducibility and possible techniques for improving the trustworthiness of scientific publications. One promising approach for addressing some of these factors is the widespread adoption of standardized procedures for experiment design, record keeping, and publication, supported wherever possible by software tools for automation and research life cycle management. This chapter identifies the functional requirements of a software system for designing and executing complex experiments and organizing their results and justifies the need for a next-generation collaborative lab information management system. We briefly describe the electrochemical sensor research which produced our group's need for the software, analyze the tooling requirements of our project, and then explore the existing ecosystem of software tools for automation and curation of research.

2.1 Use case: Electrochemical sensor arrays

Our development of a next-generation collaborative LIMS is motivated by a concrete research task, namely characterization and design of electrochemical sensor arrays for precise concentration estimation of a broad range of chemical targets [42, 71, 72]. Electrochemical sensors are sensitive to a number of interacting environmental conditions such as temperature, humidity, ambient airflow, and presence of trace interferent chemicals [44]. The sensitivity of a given sensor to a particular analyte compound is also a complex function of device geometry, electrolyte and substrate materials, and applied electrical stimulus. In order to make measurements meaningful, as much of this secondary information as possible must be collated with the raw electrical output of the sensors. Additionally, a typical characterization experiment involves a sequence of manipulations of controllable parameters such as the flow rates of input gases or applied voltage waveforms. The end engineering goal of these experiments is to determine the inverse function mapping each sensor's instantaneous output current, input voltage, and observable environmental parameters into a concentration profile of the device's chemical environment. For an experimental data set to afford such an analysis, the input conditions should be controlled as accurately as possible, and especially for batteries of tests involving many sensors operating in tandem it is necessary to employ computer control to achieve uniform results.

This experimental scenario, depicted schematically in Figure 2.1, will serve as a running example to demonstrate the capabilities and requirements of the software tool described by this thesis. Our ideal experimental setup involves commercial lab equipment as well as custom data acquisition hardware, simultaneous operation of many sensors with different physical characteristics, and precisely timed computer choreography of electrical interrogation protocols and gas flow rates. Furthermore, the exact nature of the experiments being run changes frequently as researchers identify new questions, design new sensors, and involve new equipment in their work, requiring our control and data management software to grow with the changing requirements of its users.

We believe that a software framework capable of scheduling and autonomously executing experiments of this level of complexity has the potential to be more broadly useful in any scientific environment with similar workflow needs. By generalizing our design from this use case, we hope to meet our project's needs and simultaneously provide the research community with powerful, much-needed open-source solutions for a set of problems that recur in many different scientific areas. The design goals of our software package are enumerated in the following section, followed by an overview of the existing tools which fulfill some of these requirements.

2.2 Requirements and Terminology

Laboratory science presents a diverse set of operations management and informatics challenges, and many research reproducibility efforts stand to benefit from carefully designed software tools. In this section we consider some of the many scalability challenges faced by a typical research

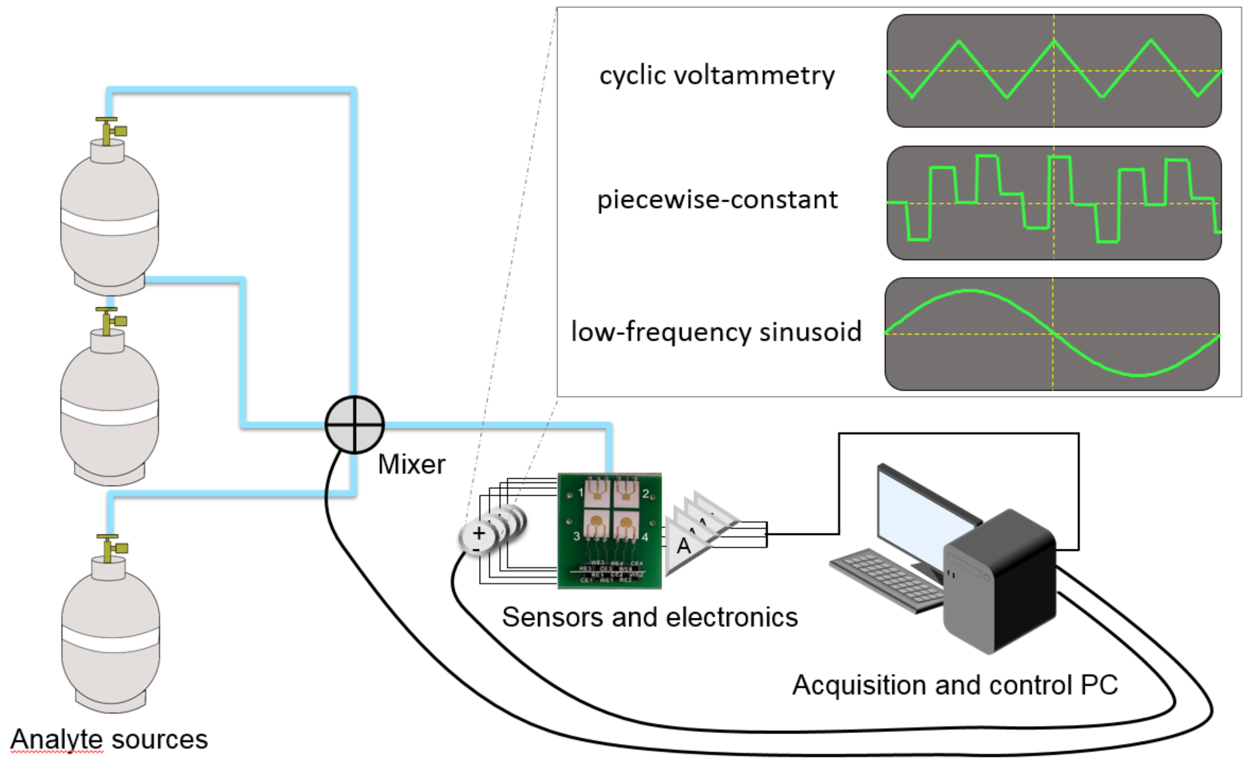


Figure 2.1: **Electrochemical sensor characterization.** Schematic of an example experimental apparatus for characterizing an array of electrochemical gas sensors. Inset: some commonly used stimulus waveforms for interrogating electrochemical sensors.

group and examine some proposed techniques for addressing them. This discussion has guided the design of the software framework presented later in this thesis.

2.2.1 Automation

The desire to scale experiments to much higher throughput provides a major motivation for exploring software solutions for lab management. As researchers begin to work with many devices and control parameters simultaneously, data collection and tracking tasks become difficult to manage. Additionally, when attempting to provide confident analyses of large sensor characterization data sets, signal processing experts require accurate information about the timing of input and output events, and adequate resolution of control events is extremely difficult to obtain under manual operation.

By employing computer control of actuators and data collection equipment whenever possible, researchers should be able to maximize the consistency of their results while simultaneously improving their productivity. The ability to automatically re-run a task with modified parameters overnight rather than carefully manipulating control dials for hours on end would allow scientists to focus their expertise on identifying new research questions rather than on tedious and meticulous experiment execution. An ideal software tool for lab automation should allow investigators to design, refine, and compose executable tasks, enabling researchers to build complex experimental protocols from a library of reusable components.

2.2.2 Metadata and data provenance

Much of the data that is collected and exchanged by researchers is stored in ad-hoc file formats, often detached from the relevant metadata necessary to make these results meaningful. Examples of metadata which are often omitted from raw data sets include measurement units, input conditions, sample and equipment IDs, and annotations such as the hypothesis of an experiment or where to find further documentation or references. These key pieces of information are often recorded or remembered only by the original experimenter and may easily become unavailable to future researchers. Even when data collection and management policies are established within a group, it requires careful discipline to enforce these rules manually, especially in a typical fast-paced research environment with little direct oversight.

Furthermore, in many cases drawing conclusions about a data set relies on information about experimental conditions that is difficult to acquire for every trial and is not obviously relevant at the outset, forcing researchers to backtrack and repeat work in order to be confident in their results. By using software to collect and manage information about the flow of data through an experiment, users can be provided with powerful tools for examining their workflows at many levels of detail without requiring costly and time-consuming repeat trials.

Systematically tracking and organizing the history of data sets as they are collected, reformatted, and undergo transformations and analysis is the focus of the growing area of data provenance [12]. Provenance techniques aim to allow researchers to properly attribute a data set, understand how it was created, and determine where and how modifications or errors were introduced. Capturing and serializing accurate and sufficient provenance information about a system remains a research topic of its own [15], but a number of existing scientific software tools provide some features that cater to this need.

2.2.3 Version control

Whenever software provides the ability to create and modify complex documents or artifacts, version control is a valuable feature for improving productivity and auditability. Similar in concept to data provenance, a version control system (VCS) keeps checkpoints of important points in a file's edit history, allowing authors to review past states, recover lost work, and make changes to a single file rather than attempting to manually keep track of backups. Version control tools are indispensable in the software industry for tracking source code, where popular tools include Git [14] and Subversion [57], but some version control features are now commonplace in office programs such as Microsoft Word's "Track Changes" mode [49]. Existing version control software for plain text files is extremely mature, full-featured, and powerful, and may be used as a third-party tool for any work where plain text code and configuration files are artifacts of interest.

2.2.4 Collaboration

Modern research labs are increasingly interdisciplinary and rely on remote sharing of techniques, data, and publications. Software designed for assisting researchers with performing and documenting their work should reflect these realities, ideally offering native support for sharing and collaboratively reviewing resources over the Internet. Software systems with distribution in mind are also well equipped to enforce policies about data usage and to maintain end-to-end provenance information about artifacts by managing records in a server-side database. Furthermore, the use of electronic media enables users to assemble information-rich dissemination units, and software which supports portable and information-dense file formats provides benefits for long-term collaboration as well as publication.

2.2.5 Extensibility

A common user complaint about commercial software with proprietary code bases is that tools are overly rigid and ill-suited for adapting to the rapidly changing needs of users [52]. The fast pace and necessary interaction with bleeding-edge technologies provides one possible reason for the proliferation of lab management software packages with slightly different goals and feature sets. To address this problem, we feel that researchers should be allowed and encouraged to customize and modify their lab management software to meet their needs. Open-source projects are theoretically arbitrarily extensible, since users may directly modify the software, but in many cases open source tools are still not designed with customization in mind. Systems with a modular design that support community-crafted plugins, user-level scripting, and straightforward integration with third-party tools are able to grow alongside users' changing needs and allow dedicated users to compound the initial learning investment over time. Such systems, when well-designed, often benefit from greater longevity and feature-richness than traditional monolithic programs [31, 47].

2.2.6 User compliance

A known challenge faced when developing software for applications such as reproducible research is that feature-rich tools often present users with a substantial learning curve, deterring widespread adoption. Tools which do not confer an obvious advantage immediately or disrupt users' existing workflows are likely to go unused, wasting development effort. Research on the topic suggests that ease of use and accessibility of documentation are important concerns for promoting user adoption [39]. Usability can also be improved and demonstrated by providing concrete examples of how the software can solve problems faced routinely by domain scientists and encouraging users to tailor the tools to their unique preferences and needs. Other important determinants of user compliance include upgradability, technical support, reliability, and compatibility with existing tools [70]. Addressing user experience concerns from the outset of a design and incorporating feedback in the development process can result in an ultimately richer product, and this is one of the key insights of the now-popular Agile development methodology [7].

2.2.7 Security

Intellectual property is an important issue in both industrial and academic research, given that funding, commercial competitiveness, and legal and professional recognition are often contingent on scientific priority. Internet-connected software which manages potentially sensitive data and design documents must therefore make digital security a principal concern. An architecture for online experiment analysis and design must carefully conform to the latest security best practices and maintain careful access controls while allowing for collaboration.

2.3 Review of existing experiment management software

The complex needs of modern research have created a large specialized software market, and there are now dozens of tools for computerizing various laboratory management and research tasks. There are now many companies offering lab informatics software with a broad range of capabilities. Since many of these programs are proprietary, it is difficult to compare their feature sets precisely, and many packages are defunct or poorly documented. Below we attempt to provide a broad overview of the major classes of software most aligned with our goals, giving a few examples of prominent products in each category. A comparison of these categories of tools and the functions they provide is given in Table 2.1.

	ELN	WMS	LIMS	Model-based design
Process specification	✓	✓	✓	✓
Analysis and documentation	✓			
Data management		✓	✓	✓
Collaboration	✓	✓	✓	
Hardware control				✓

Table 2.1: **Comparison of lab informatics software.** A comparison of the features typical of each major category of scientific informatics software. The toolkit described in this thesis intends to provide all five of the listed capabilities. .

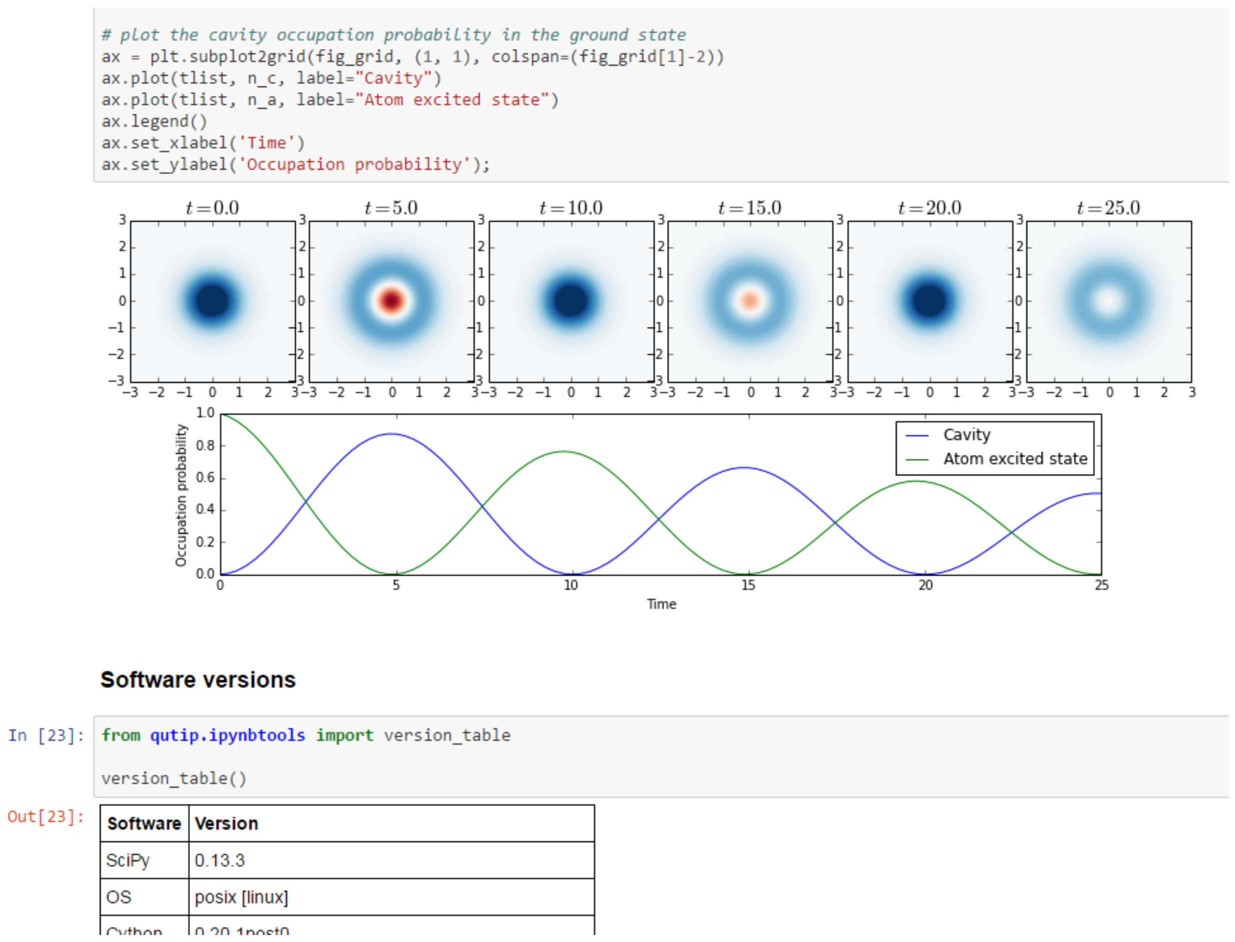
2.3.1 Electronic lab notebooks

An electronic lab notebook (ELN) is a software tool for helping researchers to chronicle their day-to-day investigations and results. A typical ELN package allows researchers to compose rich-text documents consisting of text and figures alongside technical artifacts such as data tables. Several surveys of commercially available ELNs have been published [61, 22], but the domain is still evolving rapidly and some of these programs have begun to integrate complex capabilities such as version control, experiment specification, and more. Many of the commercial products in this domain offer users compliance with the FDA's recommendation on electronic record keeping [26], a set of guidelines promoting thorough, auditable documentation of research performed in the agricultural and health sectors.

Most general-purpose programming language environments targeted toward scientific computing now include some degree of ELN functionality. These tools are typically environments for literate programming [37] which are able to embed plots and data tables alongside code and natural language documentation. Popular solutions in this domain include Mathematica [75], R [58], IPython/Jupyter [55], and MATLAB Notebook [46].

2.3.2 Workflow design tools

Defining, composing, and documenting complex procedures is a core organizational need of many research groups. A number of so-called WMSs have emerged to help manage task schedules and



```
# plot the cavity occupation probability in the ground state
ax = plt.subplot2grid(fig_grid, (1, 1), colspan=(fig_grid[1]-2))
ax.plot(tlist, n_c, label="Cavity")
ax.plot(tlist, n_a, label="Atom excited state")
ax.legend()
ax.set_xlabel('Time')
ax.set_ylabel('Occupation probability');
```

Software versions

```
In [23]: from qutip.ipynbtools import version_table

         version_table()
```

Out[23]:

Software	Version
SciPy	0.13.3
OS	posix [linux]
Cython	0.20.1post0

Figure 2.2: **Editing an IPython notebook.** Screenshot of an electronic lab notebook page in IPython/Jupyter v4.1.0 [55] integrating documentation, code, inline math, and figures.

dependencies in domains such as manufacturing [3], high performance computing [28], and business management [13]. Workflow editors provide users with a means of constructing executable tasks by describing how data moves through them, typically by visually manipulating a directed graph of processes as in Figure 2.3. In some cases workflows may serve purely as documentation, while workflow tools for *in-silico* science are often executable and may be bundled with data to provide direct replication of analysis flows on other machines. The most prominent examples of workflow software targeted toward scientists are built to facilitate the design and execution of high performance computing simulations such as Apache Taverna [53] and VisTrails [28]. Less attention has paid to scientific processes that are not completely digital and are therefore harder to fully automate. The application of similar software to managing business processes and software development suggests that these tools may also be valuable aids for describing complicated scientific experiments, and some LIMS packages provide some of this functionality [19]. By combining these workflow specification tools with software for controlling lab equipment, it may be possible to provide domain scientists with a powerful framework for defining executable specifications of complicated laboratory procedures.

A related class of software reproducibility tools encourages users to bundle input sets and sequences of data-transforming programs into a single distributable file intended to accompany published results. A notable example is [16], which uses virtual machines to produce self-contained computing environments for reproducing digital analysis under identical conditions on different physical computers. ReproZip automatically determines all the files necessary for replicating an *in-silico* workflow by monitoring the operating system during ordinary task execution. These techniques offer a promising strategy for improving scientists' ability to capture the intricacies of their work for later review or reuse while avoiding excessive demands on the user's discipline.

Industry groups have also made several attempts to produce standardized data models for business processes and equipment, perhaps the most popular of which is Business Process Model Notation (BPMN) [3], typically represented by a directed graph or flowchart much like the data models used in scientific workflow software. The most full-featured model expanding on this concept is ISO 15926 [73]. This model promises a level of generality that is sufficient to enable interoperability between businesses in different sectors and countries which rely on large, varied sets of equipment and software. The still-growing specification encompasses information as diverse as process specification and refinement, structural description of organizations and devices, component life cycle information and more. ISO 15926's representation format is based on semantic web technologies such as OWL, which employs a graph model to describe semantic relationships between entities, where each entity and relationship has an associated hyperlink. The standard has been under development for 25 years, but many specification documents have yet to be published and no software implementations are currently freely available. The extreme complexity of the

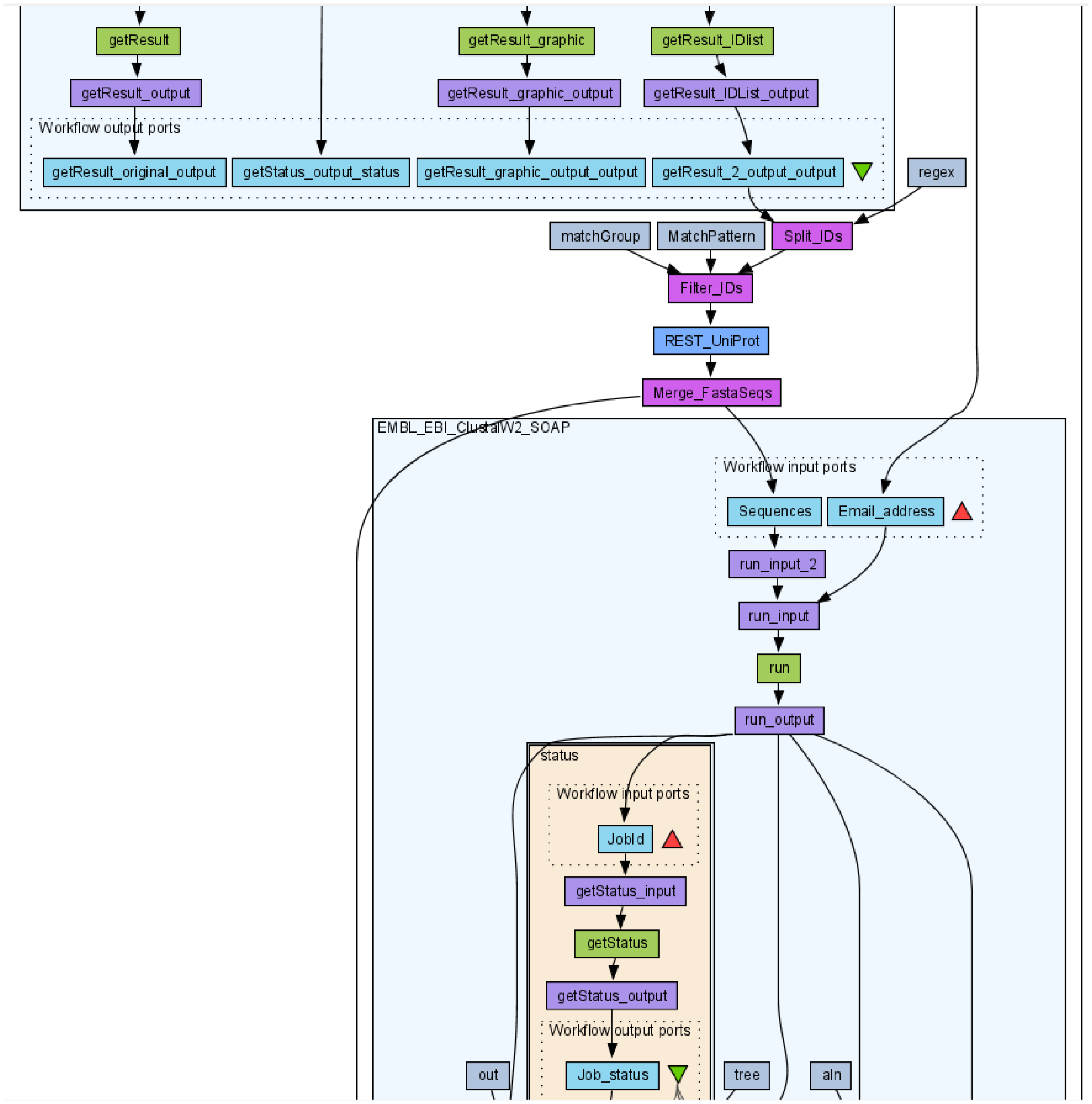


Figure 2.3: **Editing a Taverna workflow.** Screenshot of a protein sequence analysis workflow [76] being edited in Apache Taverna v2.5 [53], an open-source workflow management tool.

model is also an impediment to adoption by end users as well as implementation.

2.3.3 Laboratory information management systems (LIMS)

A LIMS is a tool for tracking the operations and assets of a laboratory. Commercial tools by this name provide a wide range of features targeted toward different aspects of an enterprise-level industrial lab such as letting researchers monitor their ongoing experiments, logging samples and data sets, and notifying relevant personnel when maintenance tasks like restocking need their attention. This field is now occupied by a staggering number of application vendors and products with a broad range of specializations, feature sets, maturity levels, and price tags [43]. These packages range from general-purpose systems built around a wiki or spreadsheet tool to specialized systems for interacting with specific types of chemical analysis equipment. Some LIMS packages provide a workflow management system (WMS) and many of them contain built-in ELNs.

The primary players in this application domain target the needs of labs in the healthcare, forensics, and pharmaceutical sectors and are mostly designed for managing and optimizing huge batch processes on fixed, well-defined equipment pipelines. The designs resulting from these assumptions would seem to make many of these programs a poor fit for the rapidly evolving experimental workflow seen in academic sensor engineering, though there are exceptions. In particular, Agilent's OpenLAB suite (formerly Kalabie) [2] offers a notebook tool which combines data collection, storage, analysis, and collaboration capabilities. This package is also capable of integrating with data collected from instruments manufactured by Agilent and some of its business partners. The tool appears to provide many of the capabilities found in a typical LIMS combined with some support for real-time hardware control, making it an attractive candidate for meeting several of our application's needs. Unfortunately, this tool is restricted to a specific set of associated hardware and at the time of this writing lacks desirable features such as modularity, user-customizability, and version control.

Most LIMS toolkits are proprietary and closed-source, but given the demand for this type of application from large industrial groups the field is in some ways fairly mature. Some of the architectural decisions that are commonplace in modern LIMS, especially their cloud-oriented model, support for user customization, and focus on auditability, seem well-suited for the kind of end-to-end research management system we intend to build. Throughout this thesis we refer to the software tool we are interested in building as a LIMS due to the broad range of functionality seen in tools which label themselves in this way.

2.3.4 Equipment automation tools

To extend automation of scientific processes beyond the purely computational domain, several vendors offer tools for coordinating simultaneous operation of actuators and data acquisition modules. Likely the most visible software package providing this functionality is National Instruments LabVIEW [23], as well as similar tools for model-based design such as Simulink [62] LabVIEW's G visual programming language allows users to connect devices, signal processing blocks, and graphical interface elements, ultimately building a custom front panel and controller for a "virtual instrument" (VI) which may communicate with many different pieces of lab equipment. LabVIEW interacts with National Instruments' line of data acquisition and control hardware and also ships with a large library of drivers for scientific instruments produced by many vendors. G programs can be regarded to some degree as workflow-style executable process specifications, but different versions of LabVIEW have well-documented compatibility problems, preventing VIs from serving as self-contained process dissemination units.

Other software toolkits have begun to capitalize on the recent emergence of affordable network-connected microcontrollers and single-board computers. One toolkit overlapping with some of our application requirements, ZettaJS, intends to provide a hardware abstraction layer for controlling and coordinating embedded data acquisition platforms over the web [77], with the stated goal of connecting devices to the IoT using existing web technologies.

Unfortunately, relatively few LIMS vendors incorporate equipment automation into their feature sets. Even fewer packages seem to recognize the ways ELN capabilities could be complemented by end-to-end experiment design and execution support. We feel that there is a promising niche for software synthesizing the best features of automation software, cloud-based LIMS, and metadata-rich ELN, and this thesis intends to articulate the design of such a framework.

2.4 Enhancing publication value

To help manage the complexities of modern research and promote reproducible science, some commentators have identified a need for more richly structured publication units than currently exist [6]. Simple examples of rich publications include PDFs containing hyperlinks to external papers or other scientific resources, and these have already begun to proliferate now that most research is exchanged digitally. Especially in scientific computing it is also desirable for publication to be executable, unifying code and documentation and allowing for complete reproducibility of a unit of research. Knowledge engineering and data archiving researchers have proposed several approaches for representing the broad space of research-relevant information in a machine-readable form, and some of these models are recognized in this section.

2.4.1 Semantic provenance models

Academic work on structured representations of research artifacts, their relationships, and their provenance has largely built on semantic web technologies such as the so-called linked data network [10]. The semantic web refers to a body of Internet resources which are connected to one another by hyperlinks which stand for specific kinds of relationships, such as subclassOf or wasDerivedFrom. Similarly, linked data are scientific resources on the web which include hyperlinks to semantically related external pages. In particular, this provides a mechanism for published data sets to record their provenance by explicitly stating a chain of relationships to their point of creation. A standards-track recommendation endorsed by the the World Wide Web Consortium known as W3C PROV [50] has recently been developed to specify how dissemination units should identify their influences. Semantic web technology has enjoyed many years of academic development and resulted in some promising high-profile projects such as DBpedia [40]. However, criticism of the semantic web's vision and approach has been readily available throughout its long history [45], and in some ways the tooling support for integrating modern web apps with resource description framework (RDF) metadata remains limited.

An important specification tool used in the linked data community is the notion of an ontology language. These are sets of RDF predicates which provide a well-defined vocabulary for discussing abstract relationships between entities, allowing domain experts to encode contextual information about concepts and resources in their field in a machine-readable format. W3C PROV, for instance, extends more generic ontology languages such as OWL2 [54] to include terms which are explicitly concerned with information about an entity's relationship to its predecessors. These "knowledge graphs" can then be examined and searched for relationships by using a specialized graph query language such as SPARQL [56]. Unfortunately, existing knowledge databases require complex external tools to draw inferences based on the semantic values of the predicates in question, such as concluding from the knowledge that A relies on B and B relies on C that A relies on C indirectly. This can make knowledge graphs very difficult to work with. Some interesting theoretical work has recently made efforts to address these shortcomings by applying mathematical tools for automated reasoning to knowledge representation, e.g. [64]. Semantic web technologies provide some interesting ideas for organizing documents and providing users with meaningful connections between research artifacts of interest.

2.4.2 Research objects

A research object is a proposed format for archiving scientific data as well as an example of a richly annotated electronic publication format [6]. The progenitors of this model argue that paper publications are inadequate to capture the intricacies of modern research activities which draw on a heterogeneous mixture of digital and physical resources. Instead, these authors call for scientists to use recent developments in social network technology and information capture to collaboratively create and share rich digital science resources such as executable workflows and electronic lab notebooks. This vision has provided a major source of motivation for our present work: we aim to build an e-laboratory software framework where research objects are native and first-class, and scientists may construct, review, and refine their experiments and analyses in a flexible, provenance-aware toolkit.

Many of the existing publications on the research object model [18, 9] use linked data and semantic web technology to specify the format a research object should use for encoding relationships between the constituent artifacts of a research object as well as between data sets and the resources that produced them. This approach seems like an interesting way to leverage the existing organization mechanimss of the linked data web to further promote the usefulness of rich publication units.

2.5 Summary

A number of software tools for automating data collection, analyzing and comparing data sets, and interdisciplinary scientific collaboration have emerged in recent years. Many of these packages provide much-needed informatics capabilities that are currently being leveraged by both academic and industry labs, especially in the biomedical and healthcare sectors. However, addressing the full set of challenges posed by interdisciplinary high-throughput sensor research and development will require the integration of LIMS functionality, an electronic notebook editor, and a scripting or graphical programming solution for equipment automation into a cloud-based software framework that currently does not exist. The remainder of this thesis will describe the proposed design and prototype implementation of a suite of software tools which synthesizes and expands upon the programs described above.

The following sections describe the design and implementation of eGor, a lab informatics software package intended to cover most of the use cases of the tools reviewed in this section and more. eGor comprises several programs which in practice typically run on several different machines and help to manage the complete process of developing a scientific experiment. This custom tool intends to provide all of the major functions listed in Table 2.1 and improve both user productivity and research reproducibility by synthesizing these capabilities into a single software tool.

CHAPTER 3
ARCHITECTURE

Characterization and development of sensor arrays presents a broad range of research challenges, not least of which relate to data organization. A LIMS adequate to the needs of our example application must provide a number of interacting software components to mediate between users and target resources such as data stores, richly featured research documents, computer-controllable lab equipment, and collaborators. This chapter abstractly describes the constituent components of the software framework we have built for collaborative design, execution, and analysis of experiments. For ease of reference we refer to our software by its pseudonym "eGor", the Digital Lab Assistant. When describing each element, we document some of the phases of our iterative design process that led to these decisions.

3.1 Network architecture

Given that the resources of interest to our software system are inherently distributed, a careful design of the system's network interconnect is critical to its scalability, security, and usefulness. Below we describe the physical system constraints driving some of our design decisions and explain how we iteratively arrived at our final design.

3.1.1 Physical architecture

Typical workflows for interdisciplinary digital research involve a number of computing resources which are physically and logically separated from each other. These include (i) individual workstations where researchers perform analysis and compose code and documentation, (ii) online information banks such as chemical and biological databases, (iii) intranet and cloud storage drives for archiving and sharing documents and data, (iv) logs of research-relevant communications such as email correspondence, and (v) dedicated, typically shared scientific resources such as lab instruments and high-performance computers. In many cases, especially in electrical engineering, a device we generically categorize as a piece of "lab equipment" is a focus of research in its own right, and can be further decomposed to include computer controllers, instrumentation electronics, and physical processes or devices of interest. Often some or all of these resources interact with each other in an ad-hoc fashion manually facilitated by users. We believe that tremendous gains can be made for research organization, accuracy, and reproducibility by coordinating the interactions between these components with a carefully designed software framework. A schematic diagram of some of these interacting components is depicted in Figure 3.1.

The most important goal of the present work is to automatically execute physical experiments by employing computer control, automatically collating raw experimental data with secondary data and metadata to produce self-contained research artifacts that are more amenable to unambiguous analysis than present ad-hoc formats. Ideally we would like for collaborating researchers at different universities to be able to review each others' experiments in real time, allowing for continuous feedback between investigators with different areas of expertise.

Although some pieces of modern lab equipment possess network interfaces and can directly act as web servers in their own right, a majority of scientific instruments of interest operate over short-range or legacy communication links. In order to allow users to remotely interact with physical resources of this kind, at least one additional machine is required. This machine is typically represented by a PC physically located in a research lab and connected directly to external hardware devices over non-networked connections such as USB.

3.1.2 Monolithic approach

A traditional architecture for web application software involves a single server executable serving presentation-layer applications to clients and making database accesses on their behalf, as in Figure 3.2. In our case, the server would also mediate access to lab equipment, providing users with indirect and high-level access to these resources in much the same way as it abstracts over the database.

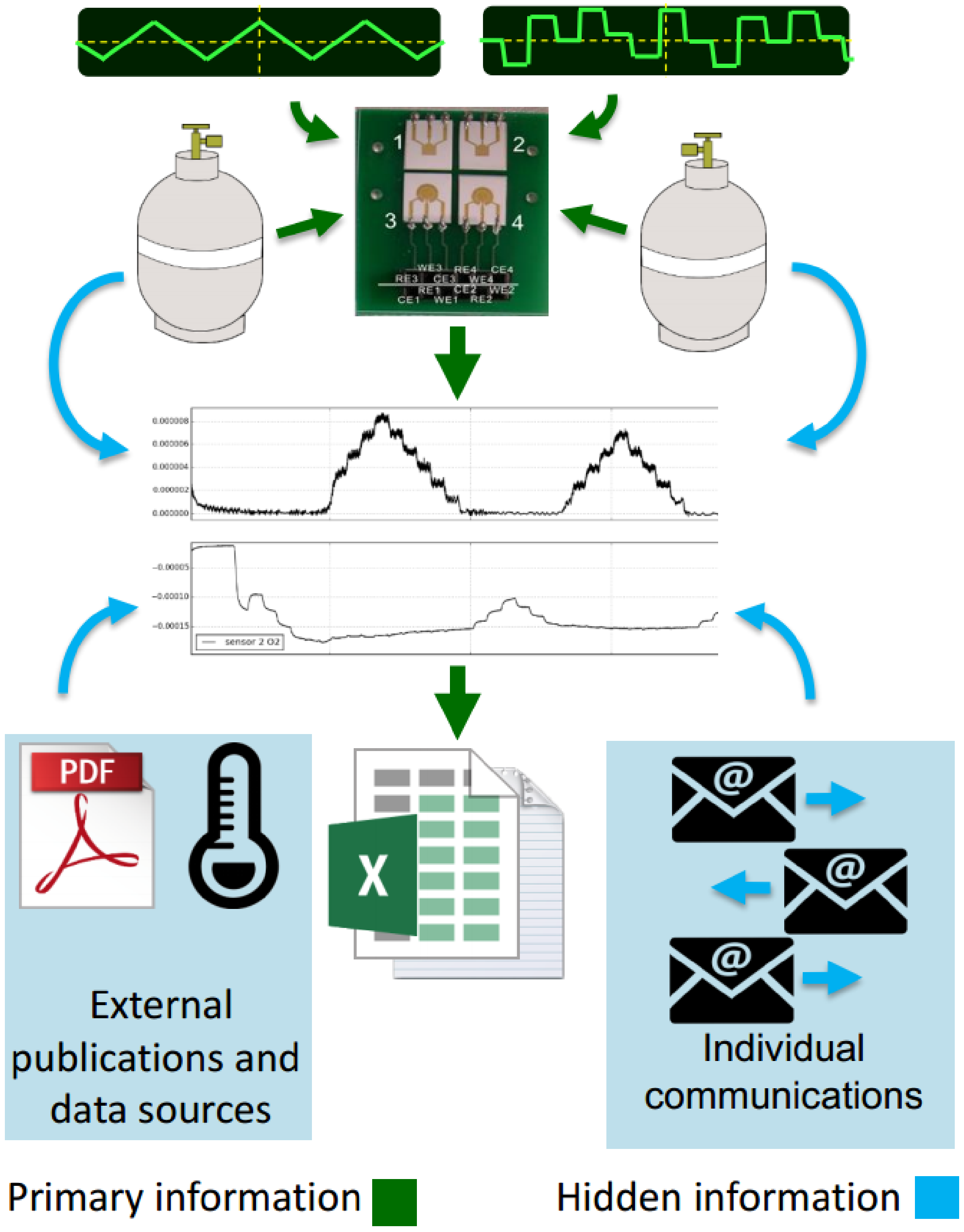


Figure 3.1: **Research-relevant artifacts.** Representation of some of the digital resources found in typical scientific workflows and their relationships. Raw data sets captured from an experimental run are often insufficient to reconstruct meaningful plots or perform detailed analysis, and researchers must rely on undocumented, hidden information sources to perform a complete analysis.

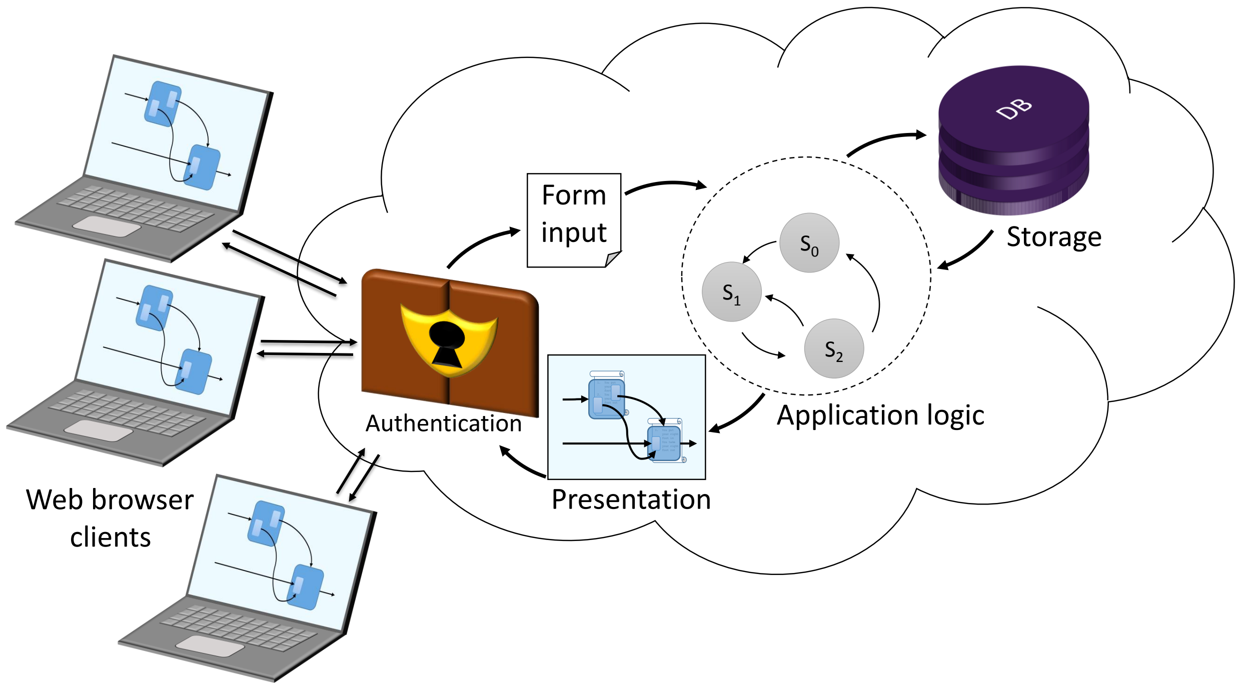


Figure 3.2: **Monolithic web architecture.** A traditional “monolithic” web application architecture where one server process manipulates a database on behalf of many clients.

This architecture is attractive for its ease of deployment and its apparent simplicity, and early in the project’s development we pursued a design along these lines. However, attempting to bundle all of eGor’s server-side functionality into a single program eventually caused difficulty with system integration. For example, coupling the code for communicating with lab instruments into the server’s application logic complicates both portions of the program and makes it difficult to test and develop them in isolation. This agrees with a common observation [65] that architectures of this kind are often less modular, making them more difficult for multiple programmers to develop independently and complicating the process of introducing new functionality. We feel that a more compartmentalized, modular approach better reflects the structure of the domain being modeled as well as conferring a number of software engineering benefits.

3.1.3 Microservices

As opposed to the conventional frontend-backend divide, some developers have suggested an architecture for web applications based on simple communicating modules termed microservices. In a traditional monolithic architecture, programmers compose a complicated application hierarchically, using one main module which calls library functions from many subordinate components. A microservice architecture splits functionality into many independent programs which communicate using ordinary network protocols, and modules are designed to assume that their dependencies are completely separate programs potentially running on other machines [41].

This approach promises better modularity than traditional web applications since capabilities can be added and extended independently of one another [5]. Since all services expose their functionality over a similar web API, implementations are decoupled from each other and internally have very different architectures tailored to their special-purpose needs. Services may even be written in completely different programming languages. The flexibility that this approach affords is a good fit with our desire to adapt the framework to meet users' changing needs. Furthermore, a microservice architecture lends itself naturally to a design where capabilities and resources are distributed geographically, as is the case with large, remotely collaborating groups of researchers. In some cases microservice architectures also scale better as performance demands on the system increase [74].

A schematic depicting the connections between some of our core microservices can be found in Figure 3.3. In our approach, no microservice is truly "central" – services may communicate with any other service provided they know its URI and present an authorized access token. Throughout the following, we use the terms microservice and service interchangeably.

3.1.4 Switchboard service

Despite its internally distributed design, the web application must present a primary gateway for user interaction. In our design this role is taken by a microservice we refer to as a switchboard,

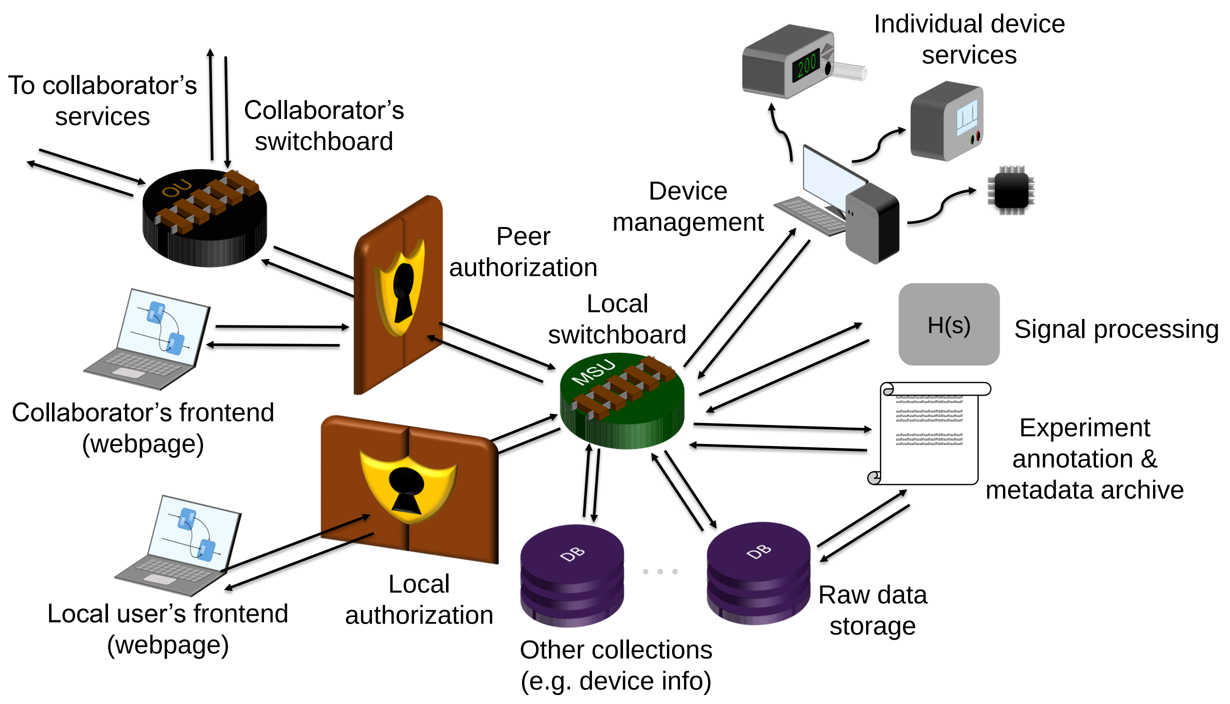

Individual device services
To collaborator's services
Collaborator's switchboard
OU
Device management
Peer authorization
Local switchboard
MSU
H(s)
Signal processing
Collaborator's frontend (webpage)
Experiment annotation & metadata archive
DB
DB
Local user's frontend (webpage)
Local authorization
Raw data storage
Other collections (e.g. device info)

Figure 3.3: **Microservice-based web architecture.** High-level interconnection between the critical microservices composing our final design.

which is primarily responsible for enumerating microservices and providing proxy access to them at appropriate uniform resource identifier (URI)s. The switchboard confirms that users are authorized to manipulate their target resources, then delegates their requests to the microservices responsible for performing actual resource accesses.

Since the switchboard is itself a microservice, multiple switchboard services may be employed by a system, affording system administrators fine-grained access controls for different components. Additionally, the switchboard of a completely different installation of the software at a different facility may be treated as an available microservice, facilitating collaboration by allowing appropriately authorized users to access external resources as if they were part of one's own installation.

3.2 Device control

A core goal of our design is to enable researchers to incorporate choreography of physical lab equipment into the executable workflows they create. Interacting with the variety of commercial and custom hardware found in a typical experimental lab requires a flexible approach, given that computer control interfaces and data formats for scientific equipment are heterogeneous and very poorly standardized. This section describes an approach for building a modular library of device drivers which integrate with the rest of the eGor framework while providing users with tools for extension and customization.

3.2.1 Instrument manager

The instrument manager is a service responsible for detecting connected devices, determining the appropriate device driver for communicating with them, and presenting a unified interface to the switchboard. This service runs as a background application on the client machine which is physically connected to lab equipment and is responsible for relaying control commands to appropriate devices as well as routing captured instrument data to sinks such as a database or real-time display viewport. Much as the switchboard service identifies other microservices and mounts them at appropriate URIs, the instrument manager identifies currently connected devices, determines an appropriate driver and communication protocol for exchanging messages with them, and exposes their high-level functionality as an API available at an appropriate endpoint, allowing the rest of the system to behave as if the instruments themselves were ordinary microservices.

3.2.2 Device enumeration

One of the instrument manager's chief responsibilities is to determine which devices are presently connected to the PC hosting the service. The process of establishing a connection with a piece of equipment and confirming its identity is dependent on the physical interface as well as device-specific packet formatting. Fortunately, many scientific instruments follow a standard convention for identifying themselves to controller PCs. In some cases, however, the instrument manager must receive explicit user guidance about which devices are connected.

Once a device produces an identification response or the user explicitly identifies an attached device, the instrument manager locates detailed device information by querying our device information service. In particular, the database record retrieved by the instrument manager includes a device driver and a protocol stack for translating low-level device commands to and from a generic high-level format. This approach allows the device-connected PC to always use the latest driver for each device, retrieve devices on demand, and communicate with any device known to a given eGor installation with minimal user interference. After the downloaded device driver code has been successfully installed, the instrument manager maps an appropriate URI to the attached device and delegates requests transmitted to the instrument to the appropriate protocol stack and device driver.

In earlier iterations of the design, the instrument manager looked for device drivers and protocol libraries in a directory on its local filesystem rather than retrieving them from the network. This would have required users to manually install or update libraries for interacting with device drivers. Additionally, the database-oriented approach allows the concrete communication code for a given instrument to be associated with the abstract data model representing the instrument as a research artifact, allowing users to examine their equipment at a finer level of detail when developing an experiment.

3.2.3 Device APIs and protocol composition

The protocol stack bundle associated with a given device is expected to expose an API that allows instruments themselves to be treated as microservices. The uniformity of this design makes it possible for the software to model many kinds of remote resources using a similar approach, and leverages existing network infrastructure to manage how commands are delegated to devices. An important responsibility of these device proxy services is translating complex sequences of commands received by the network to and from bit-level packets formatted for individual instruments. Borrowing from Internet design terminology, we refer to the sequence of data transformations and flow control operations involved in this process as a protocol stack.

To simplify and modularize the creation of communication protocols for interacting with a wide range of lab equipment, protocol stacks are designed using a library of basic data transformations as building blocks. In addition to functionally pure encoding and decoding processes, a given “layer” of a protocol stack may trigger changes in flow control or provide signals to other layers in response to certain packets. The resulting framework gives programmers the freedom to define many different kinds of communication strategies.

By compartmentalizing device drivers in this way, we improve the maintainability of the instrument management code base and provide users with the ability to extend eGor with their own modules. This is especially important for device drivers since the number of possible communication protocols is far too large to maintain an adequate library of drivers without community support.

3.3 Data model

eGor must manage data with very heterogeneous structures. In particular, research artifacts such as equipment, experimental runs, and publications may be attached to quite different sets of information. Additionally, we wish to present these records to a number of services, each of which must have access to enough information to provide a complex set of functionalities. This section outlines an object schema focused on flexibility that serves as the core model for records in our database of research artifacts. A distinguishing feature of this model is that a given artifact may have several attached groups of assets including code and data that indicate how the artifact’s attributes should be treated in different execution contexts.

3.3.1 Research artifact model

One of the most basic datatypes in our object model is referred to as an artifact, and is intended to provide a generic representation of research-relevant entities such as equipment, experiments, publications, analysis pipelines, et cetera. A given artifact is equipped with a set of “capabilities”, which are additional data records that are interpreted in different ways in different software contexts. Example capabilities a research artifact might have include a lab notebook’s ability to be edited, an experimental workflow’s ability to be executed on physical equipment, an instrument’s ability to operate as a standalone microservice, or an instrument’s ability to capture and tabulate results. Each service may optionally load some or all capabilities and interpret them in service-dependent ways to provide extended functionality.

Artifacts may also possess “assets”, which are files and resources with internal structures that are opaque to the eGor system. Examples of assets include images, code for external tools, and attachments such as PDF documents. Assets are defined by a URI and optional type information and may be accessed or created on a server’s local filesystem by services with appropriate access permissions. Using an asset rather than an object model to package data is appropriate when the data does not possess an internal structure that should be managed by eGor directly. For instance, a text file containing source code might be an appropriate choice of asset – its content may change, but eGor does not need to represent it internally as a structured object. Managing and interpreting the content of an asset is typically the purview of external tools, though operating these tools may be mediated by an eGor service. In the case of a text file, it would be more appropriate to use existing version control tools to represent the asset in a structured way.

3.3.2 Dataset management

Ordinarily data are captured via eGor-controlled lab instruments, adapted via an appropriate protocol stack, and delivered to one or more data sink services. Typical data sinks include real-time plotting and signal processing services. To support later analysis and experiment reuse, one of the core eGor features is a data tabulation service, which supports streaming live data captures into a data structure for permanent storage.

To achieve efficient usage of space and fast retrieval times, large tables of raw data are stored by a different strategy than metadata documents. To some extent these array data sets can be treated as ordinary assets belonging to an "experimental run" artifact, but datasets are special because their high-level structure must be cross-referenced with eGor artifacts encoding their metadata.

Externally generated datasets may also be added to the system by uploading known file formats, which are dispatched to appropriate adapter services and committed to the database. Similarly, previously recorded datasets may be exported and downloaded for processing with external scientific computing tools. In these situations, the user is trusted to provide the structural information needed to enrich and contextualize the raw data they enter and to appropriately document the external transformations that take place.

3.4 User experience

Each capability has a corresponding user interface component, allowing users to manipulate artifacts as well as inspect the system's inner workings from the graphical browser frontend. Using a similar mechanism to the approach described above for downloading driver code on demand, the interface plugin for a given capability is loaded when the user examines its associated artifact. Artifacts may declare some capabilities as hidden by default in order to avoid cluttering the user's workspace.

A usage example of eGor's core functionality from a user's point of view is depicted in Figure 3.4, with the following major phases indicated by numerals in the figure.

1. The user constructs a virtual workbench describing the configuration and interconnections between their lab equipment. The workbench defines the set of resources available to one or more workflows, which are specified by wiring component inputs and outputs together and providing a script of when and how to change parameters as the experiment runs.

2. The user schedules their workflow to run on the equipment during an available timeslot.

3. The workflow is compiled into a timetable of device-specific instructions. Assuming this process completes without errors, the workflow is recorded in the database as having been

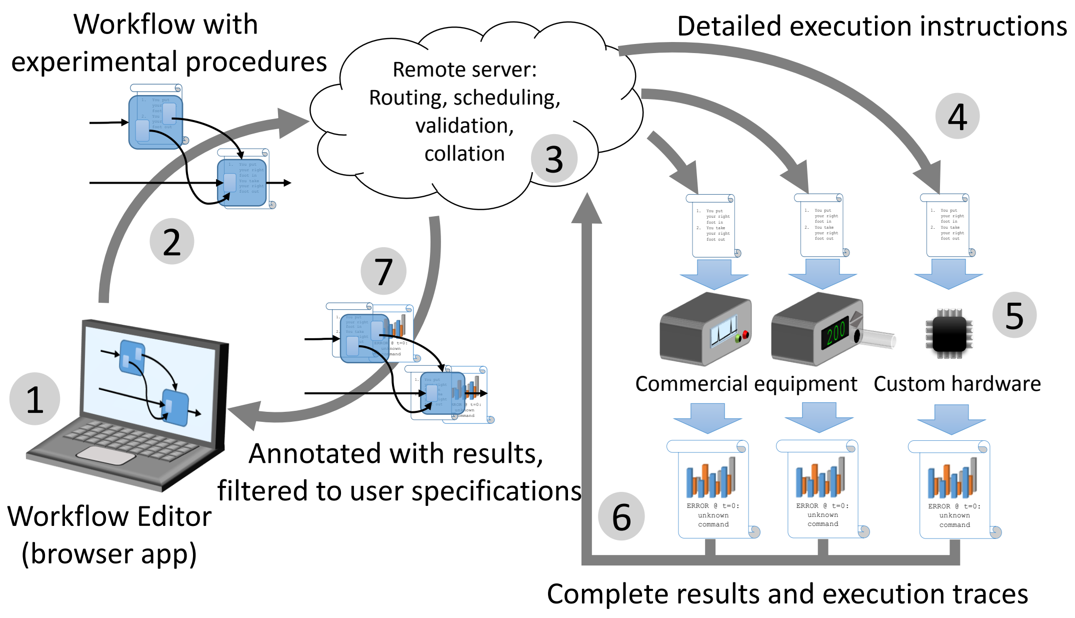

Workflow with experimental procedures
Remote server: Routing, scheduling, validation, collation
Detailed execution instructions
1
2
3
4
5
6
7
Commercial equipment
Custom hardware
Annotated with results, filtered to user specifications
Workflow Editor (browser app)
Complete results and execution traces

Figure 3.4: **A typical eGor workflow.** A typical sequence of user operations for interacting with eGor to design, schedule, execute, and analyze an experiment.

scheduled for the desired execution time.

4. The scheduled, compiled workflow is submitted to the instrument manager to await execution. Nearing the scheduled experiment time, an experiment executor service ensures that the experiment's preconditions are met.

5. The experiment executor service executes desired hardware commands at the user's specified times. As the experiment runs, real-time data is captured and streamed to the data sinks indicated in the workflow specification, allowing researchers to confirm that the experiment is proceeding as expected.

6. A complete log indicating the status of the experiment, any errors, and any failures to meet the user's constraints is returned to the server for archiving and later review. The experiment executor service attempts to confirm that experimental postconditions are met and prepares

for the next scheduled experiment.

7. The raw output log is transformed and returned to the user, producing a filtered result structure that reflects only the user's specified outputs of interest.

3.5 Security model

Especially when dealing with sensitive scientific data and remote access to expensive lab equipment, careful access control is an important architectural concern. As in a traditional client-server model, when clients authenticate themselves to the system they are provided with an access token that can be used to preserve their credentials between browser sessions. When a user makes a request through a sequence of proxy services, this access token is provided along with the request and is passed along to each service on the way to the request's destination. Each eGor service requires client services to produce an access token before it will perform work on their behalf, and a service may query a user management service with an access token to determine the identity of a user and whether their access is authorized.

Switchboard services for collaborator's installations may also query the user management service of the installation requesting access in order to either deny access to a token outright or to generate an access token corresponding to a foreign user. Users may provide labmates or collaborators with authorization rights to services and artifacts which they own or manage, and doing so modifies the list of user IDs the service will permit to access certain operations.

3.6 Summary

We have outlined an end-to-end system architecture for a set of interacting e-laboratory software components. We feel that this approach provides a reasonable combination of our target system's ambitious list of desirable features and includes a number of noteworthy design ideas. In particular, we believe that the emphasis on modularity from the ground up will be rewarded by benefits in scalability, extensibility, and user customization that are not seen in existing LIMS. In the next chapter we describe our efforts to implement this vision in detail.

CHAPTER 4

IMPLEMENTATION

The complete eGor system is a distributed application consisting of numerous software components running on several different computers. To manage this complexity, the developers have attempted to make disciplined use of best practices for web programming and to make judicious use of a range of cutting-edge third party libraries, only accepting those which have a record of stability and continued maintenance.

Our complete system draws on a wide range of technologies from every level of the software stack. This chapter provides a description of what new components were implemented by the project's development team, which external tools and libraries were used, and what architectural and practical concerns factored into the selection of these methods. This is intended to provide an overall understanding of how the application is structured rather than detailed developer documentation, which can be found at `https://github.com/egor-elab/doc`.

4.1 Overview

Figure 4.1 shows a high-level schematic of the current structure of the eGor platform. The components depicted are divided across three different machines in the simplest scenario, although more complex configurations are possible since all services interact over network-ready protocols such as HTTP. These machines are, from top to bottom (i) the end-user's PC, where an interactive single-page browser application is used to interact with various eGor services graphically, (ii) a server running microservices for functions such as authentication, remotely accessible persistent data storage, and routing requests to other services and digital resources, and (iii) one or more machines physically connected to scientific equipment of interest, responsible for managing and issuing commands to appropriate device drivers.

This chapter will elaborate the organization and communication strategies used to implement

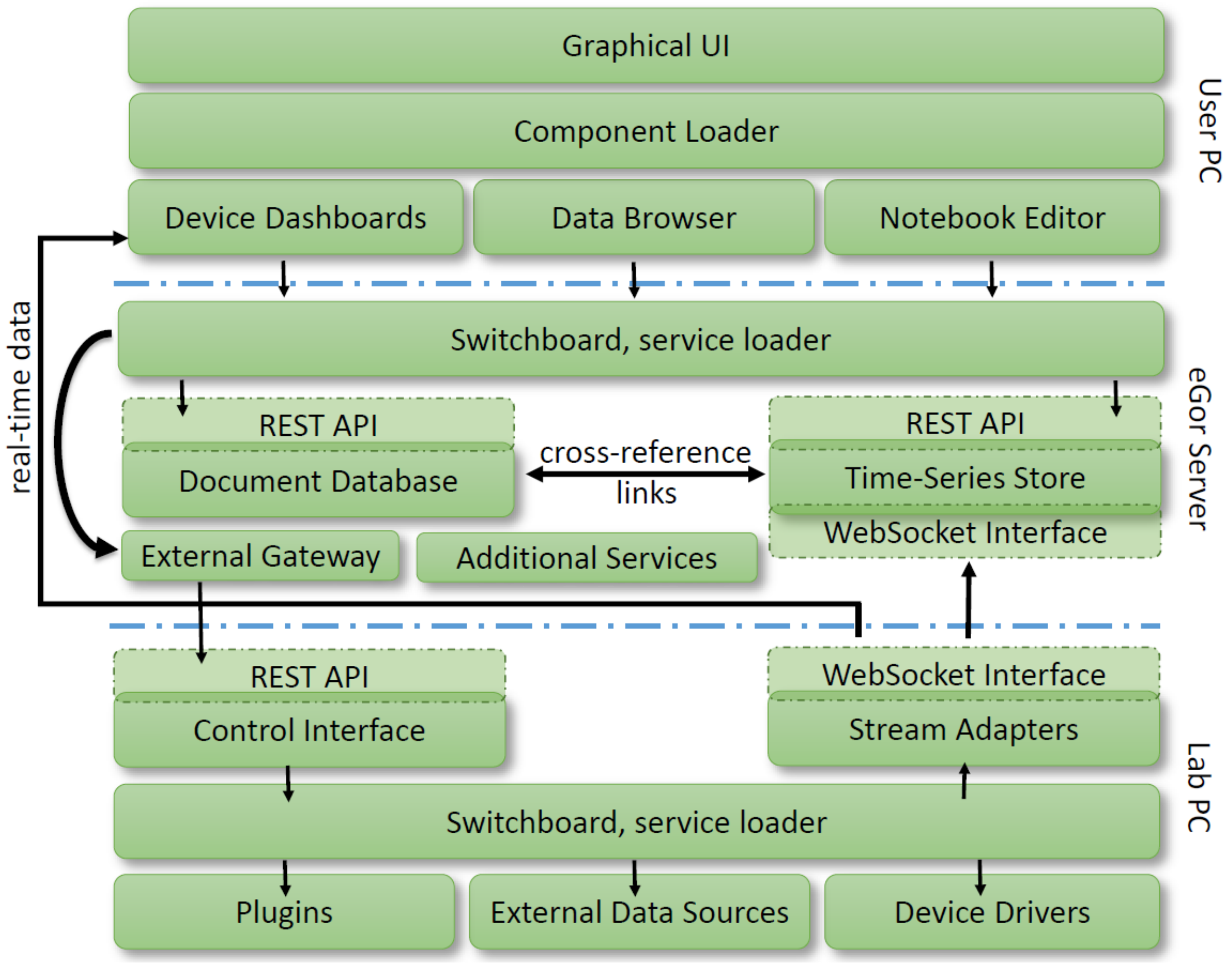


Figure 4.1: **High-level structure of the eGor system.** A high-level architectural view of the implemented eGor system, divided across three different machines where primary activity takes place: a user's machine, connected to an eGor server via a web browser, which issues commands to a lab PC running device management services to connect with lab equipment.

this framework in software, followed by a technical discussion of each component's internals. Other than the in-browser graphical interface, which is written as a single-page application in HTML5 and JavaScript using the Angular framework [32], the majority of the eGor web application is written in Python [60], making use of the mature and modern palette of networking and communications libraries available in the language. An important exception is found in some portions of the database access layer, which use NodeJS [21] libraries to present a simple and effective API. The user-facing web application structure uses all the major third party components of the popular MongoDB, ExpressJS, AngularJS, and NodeJS (MEAN) stack, but interacts with several Python microservices to add hardware connectivity.

Additionally, the eGor team has developed a model embedded device to help demonstrate how physical actuators and data acquisition modules might interact with the system – the software for this target is written in C++. This device, known as aMEASURE I, is intended to act as a flexible instrument for performing electrochemical experiments and recordings and lies outside the scope of the eGor project proper, but throughout this chapter we use it as a concrete example of the kind of device the framework supports. aMEASURE I can store and produce arbitrary waveforms by a number of methods, record digitally-converted analog measurements and stream them over a serial interface in real-time, and interact with external circuits via banks of I/O pins.

The software for all these core components is open-source and available in several Git repositories hosted at `https://github.com/egor-elab`. The diverse set of languages used helps to demonstrate a chief strength of eGor's microservice architecture: components are sufficiently decoupled that they can individually be implemented in a language and style well-suited to their unique challenges.

4.1.1 Design principles

Over the course of developing and refining the eGor toolchain, several recurring patterns have emerged which seem natural fits for addressing the application's goals and have informed subsequent iterations of the design. This section discusses several key design patterns which have been observed and employed throughout the code base, providing a feel for the philosophy of the complete system before delving into implementation details.

4.1.1.1 Request/Response vs. Publish/Subscribe

The most common form of high-level network traffic on the web is HTTP, which uses request/response exchanges to pass data between hosts. For instance, a client such as a web browser might issue an HTTP request to a server with the contents `GET /users`, causing the server to respond with a text payload encoding a resource named "`users`". The client submits user input such as form data in a similar way (typically via an HTTP `POST` action), resulting in an acknowledgment message from the server. This approach is sufficiently flexible to allow for much of the broad range of content found on the modern web, especially since servers often deliver JavaScript source code for clients to execute locally in addition to static text documents such as HTML. Issuing commands to lab equipment can often be modeled in a similar way: a controlling computer submits a configuration message and the device responds with a (possibly empty) acknowledgment that the command was received and executed. In many ways these operations are also analogous to the ubiquitous programming construct of calling a subroutine, and some authors place transactions such as HTTP actions under the umbrella of remote procedure call (RPC)s [68].

Request-and-response communication is, however, a poor fit for systems where communications must be initiated bidirectionally, with event-driven applications providing a key example. A typical data-collecting lab instrument or digital microsystem produces values in real time which must be transmitted to destinations such as databases and display monitors at roughly the same rate as they are captured. Implementations can still accommodate this dataflow into a request/response framework by periodically requesting buffers from the data source, but this approach is fraught with difficulties and is typically complicated to use when many data sources need to be managed simultaneously.

A more elegant design pattern for systems with soft real-time requirements is given by the publish/subscribe approach, also sometimes called the Observer pattern [29]. In this scheme, a "topic" or "observable" object maintains a list of "subscribers", and notifies each of them when a variable of interest changes or an event is "published" to the event stream. The publish/subscribe technique has been adopted to solve software problems like real-time data acquisition as well as for building "reactive" applications such as user interfaces and games, where graphical interfaces are expected to react seamlessly to event streams such as user input and network communications. eGor adopts this pattern for both these use cases, treating data collection devices as publishers of streams of data fragments which may be subscribed to by other services or graphical interfaces throughout the system.

4.1.1.2 Dynamic loading

One of the chief observations underpinning eGor's design is that researchers' needs are too diverse and rapidly changing to be satisfactorily addressed by a single rigidly constructed application. The implementation effort has therefore focused on constructing a core infrastructure which allows future developers to easily integrate new functionality without disturbing the system's overall operation. Each major component of eGor allows its users to load new extensions at runtime. The core subsystems each specify an interface for how a module should allow itself to be installed and expose its functionality to the network, and otherwise community-contributed extensions are not required to depend on eGor APIs or even to be written in the same programming language as the rest of the framework.

In addition to being an essential part of the daily workflow for eGor's developers, a distributed version control system (namely Git [14]) provides a runtime mechanism for achieving this dynamic loading functionality. Git was designed to allow many programmers to collaborate on a software project, share contributions remotely, and review and revert changes. Importantly, Git is distributed in the sense that each user may maintain an independent timeline of the history of the code base on a private computer, with or without network connectivity, sharing or publishing changes in a peer-to-peer fashion if and when they choose.

An example of the dynamic loading process is illustrated in Figure 4.2. The following sequence of steps describes how, for instance, the device management system might lazy-loads a device driver, waiting to download and install the appropriate code until the device has been physically connected to a given host machine for the first time. Here the phases are listed with the same numbering as in Figure 4.2.

1. A client service attempts to access functionality on another service which is not presently running, or explicitly requests for a new service to be loaded. In this case, the client is a service responsible for enumerating the serial ports available on the system and attempting to retrieve identifying information from connected devices, and its request for a device driver includes identifying information but may not name the driver explicitly.

2. The "service-hosting service", responsible for managing dynamically loaded programs, queries a database of service information to determine where it can download the requested code. The database responds with a URI specifying either a direct link to the necessary files or a Git repository.

3. The service-hoster downloads the module, possibly from an internal server or from a publicly hosted location such as GitHub [30], and executes necessary startup routines. If the service-hoster determines that the service is already loaded, it instead checks if an updated version exists and provides the client with the option to download and use the new version. eGor services are expected to implement a common interface of setup, start, stop, and cleanup scripts so that the service-hoster can install and run them automatically. The service-hoster also indicates to the new service instance how it can communicate with the system switchboard responsible for triggering the download.

4. Once the dynamically loaded device driver is live, it registers its public API with the switchboard, at which point the driver's functionality is available for other services to use.

5. The switchboard completes any pending procedure calls requested by the client service, and routes subsequent requests to the appropriate service.

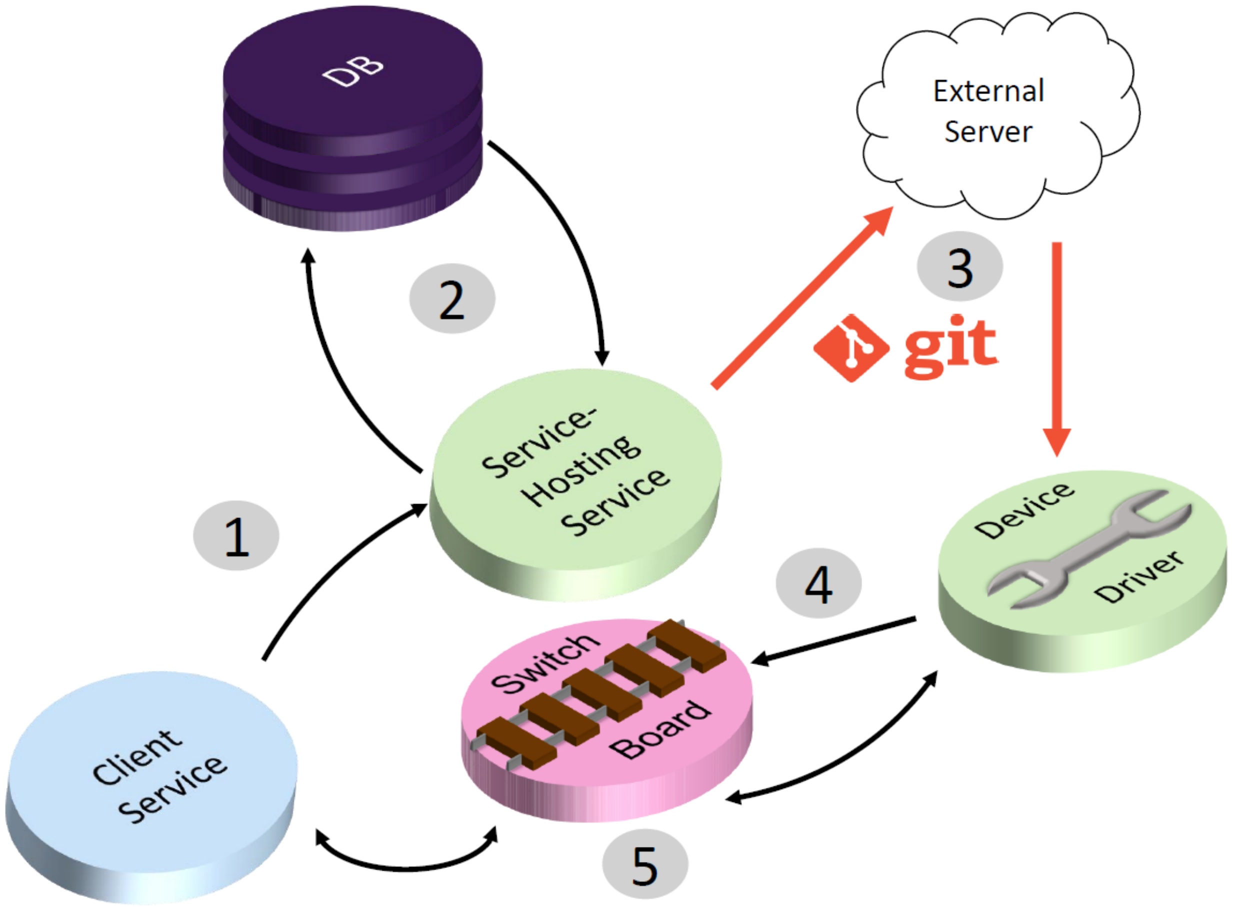


Figure 4.2: **Dynamic loading of microservices.** Phases of the dynamic loading process for downloading and installing a user-defined device driver at run-time. The service-hosting service and the services it hosts (green) run on the same physical hardware, whereas each other service may be on a different device connected via the Internet.

A similar procedure is employed for several other subsystems, such as loading new user interface components or adding new waveform generation routines to our real-time electrochemical interrogation platform.

4.2 Service interconnect

As described in Chapter 3, the microservice architectural pattern provides a strategy for compartmentalizing the development effort, promoting modularity of design, allowing for future customization and extension, and building a system that employs many different software technologies and physical machines. However, communication between microservices involves some challenges compared to traditional architectures and relies on several recently emerged web technologies to allow services to locate and use one another. Nonetheless, the microservice approach pairs well with eGor's high-level goals and has enabled us to build a flexible and sophisticated system.

4.2.1 REST APIs

One of the elements of the web's modern infrastructure that has made networked microservice-oriented applications a practical possibility is the widespread adoption by businesses and open-source software providers of relatively uniform, publicly available APIs over HTTP. Most commonly, companies expose reusable public components of their web servers as HTTP interfaces which aspire to representational state transfer (REST) principles, i.e., they model the evolution of an application's state as a sequence of transitions between states which are modeled by URIs. These conventions have allowed for unprecedented interoperability between applications written at different companies for very different purposes. As a prominent example, Google's Maps API provides a mechanism for other applications to retrieve geographical information over an Internet connection rather than maintaining independent location databases. Given that many consumers of these APIs are web browser applications which use JavaScript to issue background HTTP requests, JavaScript Object Notation (JSON) is a popular serialization format for passing data payloads to and from API endpoints. This structure, where a website provides an indexable collection of JSON-encoded resources which can be retrieved and manipulated via HTTP verbs, is often what is meant by a REST API in today's software jargon.

REST APIs are useful interfaces for making application state and data externally accessible, but are also a viable option for structuring networked communication between different parts of the same app. Most commonly, a REST API is used to provide structured database access to client code running in a browser app. For example, a news website might allow clients to retrieve a list of articles (in JSON format) by making a `GET /articles` HTTP request, then retrieve the user's selected document by querying `GET /articles/2`, then commit a submitted comment to the database with `POST /articles/2/comments`. Many of the existing tools for constructing REST APIs with web programming frameworks such as Python's Flask [4] or Express in NodeJS [20] provide for this use case.

In eGor we assign URIs in a similar hierarchical fashion, but the total application consists of a number of groups of microservices, each potentially possessing a REST API. As new services are loaded by a particular switchboard, their APIs are attached to the tree of existing URIs, much as a filesystem on a new hard drive might be mounted at a particular path on a UNIX filesystem. An eGor switchboard achieves this by serving a proxy at a URI corresponding to a known lab machine, such as `/machines/0`, relaying traffic to and from that machine which in turn provides access to connected devices as REST APIs at appropriate URIs. Information about the waveform types that a device called "aMEASURE I" is capable of producing would then be available by accessing `GET /machines/0/devices/aMEASURE_I/wave/info`, assuming that the REST API for `aMEASURE_I` understands how to interpret the path `/wave/info`. Specifying a REST API is part of a user's responsibility when defining a driver for a new device, as explained further in section 4.5.2.

4.2.2 WAMP routing

Web Application Messaging Protocol (WAMP) is an open protocol and software stack definition created by Tavendo, who provide reference implementations in several languages in the form of the Autobahn protocol libraries and a request router called crossbar.io [67]. The authors of these tools claim that their protocol simultaneously addresses many of the use cases of existing protocols for machine-to-machine communication such as Advanced Message Queueing Protocol (AMQP) and socket.io [59]. The protocol is built on top of WebSockets, which uses a TCP connection to achieve reliable full-duplex streaming and is now supported by all major web browsers and a number of web frameworks in several programming languages. One advantage provided by this protocol design approach is that machines for hosting microservices or acting as clients for WAMP networks can require less special software than is required for using some message queuing infrastructures such as AMQP, simplifying the installation process for end users.

WAMP provides a set of capabilities which are a good match for our application, including built-in support for routing remote procedure calls between any two connected services and bidirectional publish/subscribe-style message passing [35]. Notably, WAMP also describes how a service called a router facilitates organized communication between nodes by redirecting remote procedure calls and data streams to appropriate client endpoints. The protocol was explicitly designed to simplify the implementation of IoT applications, especially those with service-oriented architectures that span multiple devices of different types. Furthermore, WAMP is designed to target many different languages and target devices, providing a common network interface between server-side code, browsers, and mobile apps, and the abstraction and separation of concerns provided by such a framework is well-matched to heterogeneous service-oriented architectures such as eGor's. This provides an attractive solution for addressing many of the problems faced when developing our system, especially given that it allows for dynamic registration and removal of remotely-callable methods, flexible routing of data sources through different machines and endpoints, and is inherently bidirectional in the sense that any service can initiate communication with any other so long as it has sufficient security privileges. In our existing implementation, WAMP's capabilities have primarily been used for service-to-service communication and to organize streaming data transactions from data sources to sinks such as real-time plots and array storage services. A more elegant and truly service-oriented design could be achieved by adopting WAMP for issuing user commands and making database accesses as well, but at the time of this writing WAMP has poorer tooling and documentation than some of its more established counterparts.

4.3 User interface

Providing a useful and manageable interface for scientific users who are not computer experts is one of eGor's most important design constraints. The development effort has leveraged several powerful libraries for developing web applications and providing desired user interface features to allow researchers to immediately take advantage of eGor's capabilities.

4.3.1 Thin client design

All graphical interface components of eGor are implemented as interactive web pages and require only a modern web browser and an Internet connection to use. In this way the user interface acts as a thin client portal connecting users to an eGor server. This approach has the advantage of requiring no installation on the user's part other than registering an account, as well as making software updates transparent to users, since the latest version is automatically retrieved from the server each time a user connects. Furthermore, our approach implements the user interface as a single-page application, meaning that a user's entire session takes place without retrieving more than one page from the server or refreshing, instead using asynchronous HTTP requests in the background and bidirectional WebSocket communication to synchronize application state with the server. Structuring client-server interactions in this way helps to decouple the browser from the back-end, allowing these components to be developed independently. By leveraging abstractions provided by WAMP's application framework, it is possible for user interface components such as plots and control panels to act as peers with other microservices, simplifying the structure of the program.

4.3.2 Angular 2

The JavaScript framework underpinning eGor's browser app user interface (UI) is Angular 2, a complete rewrite of the popular and sophisticated AngularJS framework for building single-page applications [32]. Angular gives programmers a toolkit for defining custom HTML5 tags with dynamic behavior and for "two-way data binding" between elements of a web page and JavaScript objects. This means that display elements are automatically updated when specified JavaScript variables change, and similarly user inputs such as changes to form elements are automatically reflected in bound JavaScript data structures. This synchronization between elements of a page's document object model (DOM) and corresponding variables in the JavaScript program allows for a declarative programming style to be used to describe how an app is displayed while writing the control logic with sequential, imperative JavaScript.

Advantages of Angular 2 over its predecessor and other client-side programming frameworks include a structured, object-oriented style, a focus on reactive programming using the Observer pattern, and improved support for asynchronous functionality such as lazy-loading components and application structure. In particular, the design patterns embraced by Angular allow eGor's developers to carry a modular, service-oriented philosophy through to the user interface, compartmentalizing functionality into a connected group of independent, dynamically loaded services.

4.3.3 UI components on demand

Although Angular is built for creating single-page browser applications, their associated JavaScript code often makes many behind-the-scenes network requests to retrieve requested or up-to-date information. A map application provides a familiar example: rather than loading geographical information about the entire globe when the page is first loaded, new connections to REST APIs are made asynchronously in the background to retrieve more data as the user pans and zooms the map to view different locations. In addition to providing full-featured abstractions for retrieving and managing remote data sources of this kind, Angular 2 has capabilities for dynamically loading application code as well. A typical use-case for this feature is to lazy-load components that do not need to be present in the page initially, reducing startup time by waiting to download some JavaScript modules or HTML templates until the user navigates to a state which requires them.

eGor's design makes use of this dynamic loading functionality to allow for arbitrary extensions to the UI. Each document stored in eGor's artifact database may be associated with one or more UI components, self-contained Angular modules which provide special-purpose functionality associated to a user, device, or experiment. For instance, the aMEASURE I electrochemical measurement device provides a real-time data monitor and several control panels corresponding to the different waveform generation mechanisms it supports. The database record storing information about this device contains an entry for each of these UI widgets including links to JavaScript code defining an Angular component and its business logic, HTML and CSS files declaring how they should be displayed, and links to additional asset files such as images and PDF operator's manuals.

As with other elements of the eGor framework that employ dynamic loading, this design allows the core software to remain small, efficient, and single-purpose while allowing future developers and users to create and customize components to meet their needs. By choosing WAMP as the network interconnect between services, we have also made it possible for JavaScript components in the browser to interact with back-end services over the same communication interface as the microservices use with each other, promoting scalability and modular design. This component-based approach to building the browser app could be supplemented by a graphical tool allowing users to drag-and-drop interface components to build their ideal control panel in a similar way that users of LabVIEW are accustomed to constructing control panels for their virtual instruments [23]. This approach also aligns with our microservice architecture, where functionality is contained in independent interacting components which may have very different internal behaviors. The decoupled design also allows the system to embed third-party web components and even entirely different browser apps into the front-end, enabling eGor to integrate existing open-source software packages such as electronic lab notebooks with our design.

4.3.4 Jupyter

The Jupyter project (formerly IPython) is an open-source software tool providing a flexible architecture for creating electronic lab notebooks for scientific computing. Jupyter now supports many of the most popular programming languages for science and engineering applications and has several extension packages providing additional functionality, most notably JupyterHub. JupyterHub provides a web server which allows teams to share and collaboratively edit notebooks in the browser, embedding plots, equations, code and more inside a document that doubles as an executable analysis program. This tool is well-supported and has addressed many of the important challenges associated with building a collaborative documentation tool for computational science.

eGor experiment artifact model includes a UI component called "Notebook" which embeds a Jupyter Python notebook inside the eGor browser app. This allows researchers to attach analysis and observations to an experiment in a flexible format using a full-featured language and toolkit for scientific computing. Since many eGor components were developed with Python, this also provides a mechanism for users to interact directly with other system components at many levels of abstraction, potentially interleaving instrument control commands and signal processing in a single notebook. Python libraries such as Numpy [69] or Pandas [48] also provide powerful high-level APIs for interacting with data sets, and eGor provides access to an experiment's raw data files from within the associated Jupyter notebook. This approach uses a mature program to supplement eGor with much-needed ELN functionality and demonstrates how the modular design allows for embedding useful third-party tools into the browser app.

4.4 Database management

A careful choice and implementation of the system's data model is important for performance, flexibility, and determining the organization of the app. eGor employs multiple server-side data stores for persisting datasets, images and documents, information about experiments, devices, and users, and records describing eGor components such as code for dynamically loaded modules.

4.4.1 NoSQL and schemaless databases

For many years web applications primarily used relational databases for persistent data storage, querying and assembling them using Structured Query Language (SQL). These databases are based on well-understood theoretical foundations and have a number of advantages for applications such as business operations management and traditional web architectures, but have performance difficulties when dealing with complex data structures that are not naturally suited to a table format

[51]. Relational databases are often also rigidly tied to a database schema, making them ill-suited for records with many small variations in structure or with structures that change over time.

In response to the difficulties posed by this technology, considerable investment has been made in recent years in developing alternative database styles which offer more flexibility and parallel scalability. These so-called “NoSQL” databases have native data structures such as graphs, key-value pairs, or object models such as those found in object-oriented programming languages. Often these tools bill their data models as “schemaless”, contrasting themselves with traditional relational databases where administrators must provide a predefined set of names, types, and relationships for the rows when a new table structure is created. Claimed benefits of NoSQL database software include improved adaptability to changing data formats and better performance for some applications [38].

The primary information of interest to eGor is complexly structured and chiefly concerned with relationships between publications, experiments, data sets, and devices, and we sought to adopt a database technology capable of naturally modeling these artifacts and their connections. For this reason we initially examined graph databases and relational database representations of semantic web content, but ultimately chose the document database MongoDB to reflect the nested, inheritance-focused structure of our data model. MongoDB’s internal structures also map directly to the JSON object structure used for communication and state representation throughout the eGor system. The popularity of document databases in the NodeJS ecosystem has created a thriving space of open-source tools for working with systems like MongoDB and connecting them to other important application components.

Constructing an API layer to expose RESTful access to a database involves a substantial amount of boilerplate and can become quite error-prone and difficult to manage as system and data model complexity increases. StrongLoop LoopBack [66] is an open-source framework which generates API endpoints and documentation for one or more server-side data stores. LoopBack is built on top of the NodeJS web application framework ExpressJS, allowing it to be used in conjunction with the wide array of community plugins and middlewares available for Express. In eGor, LoopBack is used to produce object models for artifacts such as instruments, virtual workbenches, experimental runs, result sets, and eGor software plugins. LoopBack generates hand-customizable REST APIs for declaratively defined data models, automatically handles accesses to several different database backends, and includes application logic for important fundamental tasks such as access controls, account creation, and file uploads.

4.4.2 HDF5

Although NoSQL databases such as MongoDB provide a flexible solution for persistent storage of complex document-like data, they are ill-suited for efficiently querying large array-like scientific data sets. HDF5 is a technology for manipulating multidimensional time-indexed formats that has seen strong adoption in the finance, machine learning, and data science fields in recent years [33]. The open source working group responsible for developing the HDF5 specification has also provided a Python web server for storing datasets and presenting them as network resources, which includes a reference implementation of a REST API for remotely manipulating and extracting subsets of datasets.

eGor keeps a record of a given experimental run by capturing its metadata in a MongoDB collection reserved for cataloging past experiments. This document contains annotations about the experiment's purpose and outcomes, links to related artifacts, such as a workbench record describing the instruments involved and their connections, and an embedded result log document providing information on the experimental results. This result log provides information about where the tables of associated data can be found, in the form of links to HDF5 resources stored elsewhere on the network, and metadata about how the attached data sets should be interpreted such as information about measurement units. The HDF5 interfacing portion of eGor also includes a microservice which subscribes to data streams published by device drivers over WebSockets, buffers the incoming data, and appends it to appropriately organized HDF5 stores.

4.5 Device management

eGor's core framework includes a microservice which runs on an instrument-connected lab PC and is responsible for detecting which instruments are connected, loading appropriate device drivers and protocol-translating software modules, and presenting a uniform network interface for handling device controls and data in the form of REST and WAMP APIs. This section discusses the design of the instrument management service (written in Python) and explains how some of the challenges in its implementation were addressed.

4.5.1 Enumeration

One complication that arises when attempting to communicate with many different lab devices is that it is difficult for a PC to determine exactly which devices are connected to it. Many scientific instruments use legacy protocols and hardware such as serial or parallel ports which provide no built-in mechanism for identifying a device or its capabilities to a host machine. To address this problem, the measurement equipment industry standardized the Virtual Instrument Software Architecture (VISA) API, which is implemented by instruments from a number of different manufacturers [27]. To determine what device is connected and load an appropriate device driver, the host must send a message to the instrument asking for identifying information.

Typically serial instruments are connected to modern PCs using USB-to-serial adapters. Further complicating the enumeration process, information provided to user-space applications about USB connected devices and USB connection events varies by operating system and does not necessarily contain information about whether a given USB device is a serial port. The eGor microservice responsible for tracking connected devices therefore uses the following algorithm to keep an up-to-date record of which equipment is connected.

1. When a USB event occurs signaling connection or disconnection of a device, the instrument manager triggers a re-scan of all serial ports.

2. To scan a given port, the instrument manager transmits the VISA identification command "`*IDN?`" and waits for a response. Since the communication rate of the target device is unknown, this step must sequence through a list of commonly used baudrates, pausing after each transmission to see if a response is received. Since some devices of interest to not conform to the VISA specification, different identification messages are transmitted to probe for some other known devices.

3. Once a response is received to an identification request, the database is queried with the instrument's response string and the baudrate that was used to retrieve it. The database is searched for a known device driver matching this profile.

4. If an appropriate device profile is found, the service-hosting service on the lab PC is asked to download and install the microservice used to manage interaction with the target device.

5. The device driver microservice is brought up, issues startup commands to the device, and registers its high-level API with the eGor switchboard.

4.5.2 Protocol stacks

One of the most important capabilities of the eGor system is its ability to connect to computer-controlled lab equipment. Scientific processes often involve a wide range of legacy instruments which use different communication protocols and data formats. Rather than attempting to provide individual drivers and protocol translators for the many devices that our future users may need, we have constructed a Python library of simple protocol building blocks which can be connected into more elaborate protocol stacks.

We abstractly define a protocol layer as a composable software unit for transforming units of data from one format to another. A protocol layer consists of a codec, a transport, and a controller. One or more of these subcomponents may inherit the default implementation from the "layer" base class, which simply passes data through unaltered. A codec is a direct, often reversible, transformation from one encoding to another, a transport is responsible for logically partitioning streams into different units, and a controller determines how packets are generated or consumed at each layer. Layers are bidirectional and symmetric by default, in the sense that device-bound traffic and host-bound traffic are assumed to have the same format at a given layer. Using different layer assembly functions provided by the library, layers are "stacked" to form more complex protocols

Since the protocol stack abstraction captures the process of transcoding data between different formats as well as managing data transfers, combining a sequence of protocol layers can be used to assemble a Python device driver for a custom or commercial instrument. A completed eGor-compatible driver captures and transmits data to a low-level byte interface at one end and presents a high-level network-connected API at the other. eGor drivers typically expose a set of remote procedure calls for manipulating the associated device and produce one or more event-driven data streams which are published to the WAMP router. A developer may make a new device driver known to the system by assembling an appropriate protocol stack, identifying its network-facing communication points, and pointing eGor's database at a Git repository containing the driver code and an appropriate metadata file via the graphical interface. We have developed complete device drivers for aMEASURE I, its successor aMEASURE II, and a commercial gas flow control device manufactured by Alicat. Additionally, the eGor team provides a "seed" repository containing skeleton code for a custom device driver at `https://github.com/egor-elab/driver-seed.git`. We are hopeful that researchers who find our toolchain useful will help support the project's long-term usability by developing and contributing to a library of device drivers and other eGor services.

4.6 Summary

This chapter outlined the design patterns employed by the implemented eGor software and investigated some of the concrete tool selections and problem solving approaches chosen for assembling the system. The resulting application draws on a broad swath of

CHAPTER 5

SUMMARY

This thesis has provided a review of the design and implementation of software systems for facilitating reproducible research and has described eGor, a toolkit for collaboratively specifying, executing, and analyzing real-world experiments. The completed design has a number of subsystems and draws inspiration from a number of existing free and commercial software packages, and would not be possible without the huge range of open source libraries available for web programming today. Nevertheless, creating eGor has involved solving a number of difficult software development problems and has resulted in several design approaches and capabilities that appear to be unique. While minimum functionality has been achieved, this project is an ongoing effort and will rely on continued contribution from core developers and the open-source community to make a truly powerful framework.

5.1 Contributions

eGor was designed in response to a set of challenges that are not adequately addressed by existing scientific software. The tool described in this thesis combines a number of features that have not previously been explored in this application domain, centered around a modern, flexible web application architecture built on a broad range of advanced software technologies and design patterns. Here we reiterate the unique features of the system and their value for automated, reproducible research as well as other large scale web applications.

5.1.1 Hardware-connected lab informatics

eGor represents the only software solution we are aware of for cloud-based structured management of experimental data and procedures that is also capable of directly interacting with arbitrary commercial or custom hardware. The system's design allows users to remotely control lab equipment via an ordinary browser interface, but also allows scheduled sequences of device commands and parameters to be captured alongside datasets, discussion, and analysis in an electronic lab notebook document. Raw datasets and the metadata necessary for their interpretation can be automatically recorded in a richly structured archival format, allowing collaborators or reviewers to examine, analyze, and share the complete lifecycle of an experiment involving manipulations of physical equipment as well as sophisticated software analysis. We believe that this integrated set of capabilities will improve researcher productivity, the reproducibility of scientific work, and the state of software-driven scientific workflows in general.

5.1.2 Modularity via microservices

The core of eGor's design is its focus on subdividing functionality into small functional subsystems called microservices which communicate over ordinary network protocols and may be composed into larger systems. This is not a design innovation on its own, and in fact is being increasingly adopted by a number of technology companies to power their internal operations. However, eGor makes extensive use of dynamic module loading to allow new microservices to be created and enabled at runtime and seamlessly integrated with existing functionality. This behavior is made possible by a special microservice responsible for installing other services. By compartmentalizing each service into its own version-control repository, ystem components can also be upgraded using this mechanism whenever they change, allowing the entire system to cooperate with continuous integration strategies and making sure all users stay up-to-date. Additionally, each device driver is encapsulated in a service, and this strategy at once simplifies the design and makes it possible for users and eGor developers to add support for new devices far into the future.

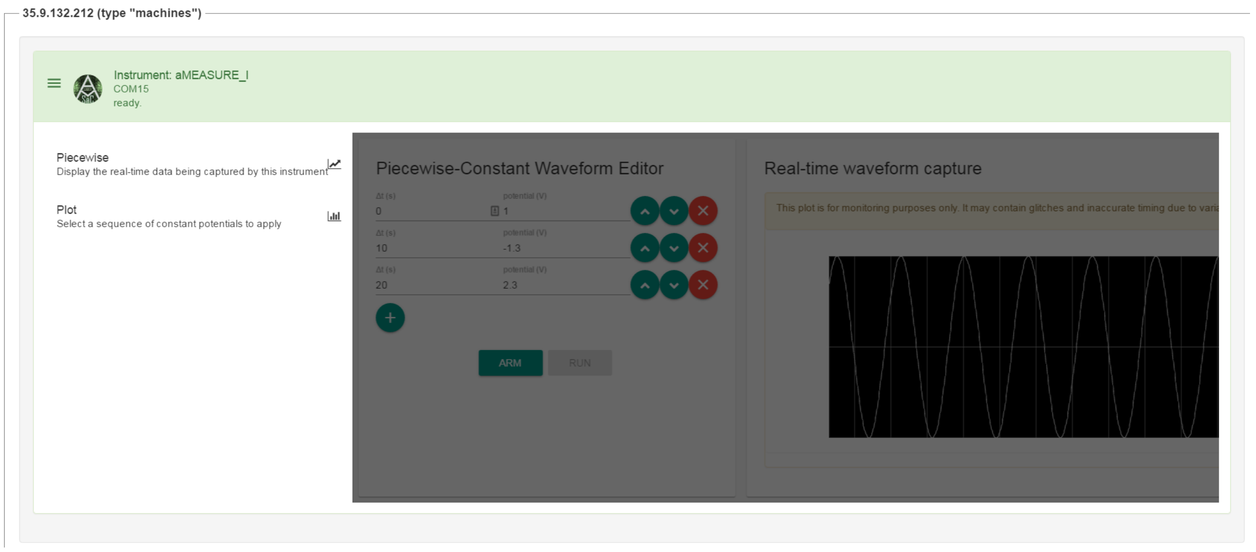
5.1.3 Design for customizability

The ability to dynamically load new services also provides users with a powerful mechanism for customizing and extending the eGor system to meet their needs. Ultimately the developers envision an ecosystem where researchers can create and share services independently, collaboratively extending eGor in much the same way as scientists share publications or technical tips. eGor's design is marked by focus on user customization, and we provide libraries for creating new device drivers and user interface components. User interface components may also be loaded dynamically into the browser interface, and each service or device driver may be associated with any number of these graphical control panels. The browser app makes use of a "hot reloading" approach to allow parts of the page to be updated without performing a complete refresh, enabling a very fast development cycle for users to create and customize their control panels. We believe the novel architectural choices made to enable these capabilities will help to overcome the limited longevity of many lab informatics software packages.

5.2 Implementation status and future work

The design vision elaborated in this thesis is ambitious and has not yet been completely realized. At the time of this writing the system's core architecture is in place, but some usability features have yet to be implemented. The infrastructure for dynamically loading microservices and communicating between them has been completed, and the system can automatically detect connected devices, look up appropriate drivers by querying our internal server, and bind the driver to the connected port. Real-time remote interaction with devices via the browser interface is available, as shown in Figure 5.1. A small set of device drivers for the equipment used by our collaborators has been implemented by composing together simple protocol and control elements from a library provided as part of the eGor system.

An archiving subsystem has also been developed, allowing data streams from multiple devices to be permanently recorded in an efficiently indexable table format. These data tables are cross-referenced with a database of user-provided metadata describing the experiment which created them, and can be examined via the user interface and downloaded for external use. This functionality is intended to allow researchers to manage their datasets alongside the specifications for their


35.9.132.212 (type "machines")
Instrument: aMEASURE_I
COM15
ready.
Piecewise
Display the real-time data being captured by this instrument
Plot
Select a sequence of constant potentials to apply
Piecewise-Constant Waveform Editor
ARM
RUN
Real-time waveform capture

Δt (s)	potential (V)
0	1
10	-1.3
20	2.3

This plot is for monitoring purposes only. It may contain glitches and inaccurate timing due to varia

Figure 5.1: **User interface screenshot.** Screenshot from the web browser front-end, showing a user interacting with a remote piece of equipment. Data may be captured and monitored in real time, and the device may be controlled either using graphical interface elements or directly communicating with the device over a browser-embedded terminal.

experiments, ultimately building publication units which include detailed links to all the research artifacts that contributed to a set of findings.

Given the scope of eGor's application domain and the opportunities such a tool provides for researcher productivity and scientific auditability, the developers have begun to maintain large and continually growing list of desired features. Much of the functionality still to be implemented has to do with improving the richness of the experiment metadata model. In particular, researchers should be able to compare different experimental trials and configurations to better understand the impact of changing a parameter or piece of equipment. A more object-oriented approach for defining and refining experiment templates would also be beneficial for improving research productivity, and an object database may be a good fit for improving how systems are modeled. Additionally, given the infrastructure already in place it should be straightforward to expand the existing system to allow interaction between many different machines and independent eGor installations, but this functionality was not necessary to our immediate use case and is not yet implemented.

5.3 Conclusion

As it exists, eGor is usable for remotely controlling devices and for capturing and sharing data in richer formats than are currently typically found in ad-hoc scientific data collection. The core architecture has been implemented for allowing user interface components, database access layers, and device drivers to interact, and due to the architectural focus on modularity and runtime extensibility we believe that future users will be able to gradually extend the system to meet their unique research goals. The current implementation acts as a usable proof of concept for the vision elaborated in this thesis, connecting researchers, equipment, and data in unprecedented ways using the nascent Internet of Things as a technological substrate.

APPENDICES

APPENDIX A: ACRONYMS

AMQP Advanced Message Queueing Protocol. 45, 46, *Glossary:* AMQP

API application programming interface. 3, 4, 27, 29, 30, 42, 44, 45, 51, 53–55, *Glossary:* API

BPMN business process model notation. 15, *Glossary:* BPMN

DOM document object model. 48, *Glossary:* DOM

ELN electronic lab notebook. 13, 17–19, 50, *Glossary:* ELN

IoT Internet of Things. 3, 4, 46, *Glossary:* IoT

JSON JavaScript Object Notation. 44, 51, *Glossary:* JSON

LIMS laboratory information management system. 2, 3, 6, 13, 15, 17–19, 21, 23, 36, *Glossary:* laboratory information management system (LIMS)

MEAN a web application software stack consisting of MongoDB (database), ExpressJS (web server middleware), AngularJS (client front-end), and NodeJS (network programming architecture). 39

RDF resource description framework. 20, *Glossary:* RDF

REST representational state transfer. 44, 51, 53, *Glossary:* REST

RPC remote procedure call. 40, *Glossary:* RPC

SQL Structured Query Language. 50, *Glossary:* SQL

UI user interface. 47–50, *Glossary:* UI

URI uniform resource identifier. 28–30, 32, 42, 44, 45, 66, *Glossary:* URI

VCS version control system. 10, *Glossary:* VCS

VISA Virtual Instrument Software Architecture. 53, 54, *Glossary:* VISA

WAMP Web Application Messaging Protocol. 45–47, 49, 53, 55, *Glossary:* WAMP

WMS workflow management system. 17, *Glossary:* WMS

APPENDIX B: GLOSSARY

AMQP a network protocol for sharing data and computations between a cluster of connected computers. Implemented most prominently by RabbitMQ [63]. 45

API An application programming interface (API) is a publically exposed set of software functionality intended to be used for composing other applications. In a web programming context, sometimes a REST API is meant. 3

artifact a generic term for an entity of interest to a research project, especially a data set, source code of a program, or an experimental protocol specification. 10, 13

backend the internal server-side logic of a web app, typically responsible for interacting with databases and performing intensive computations . 27

BPMN Business Process Model Notation, a specification of a flowchart-like format for describing procedures found in business and manufacturing settings. See [3]. 15

data provenance A generalized term for tracking scientific data as it undergoes a sequence of transformations from raw data into a publication-ready figure. 10

database schema a specification describing the structure of allowed database entries . 51

declarative a programming style which focuses on asserting relationships between software entities rather than describing the state transitions necessary to transform data . 48

design pattern a recurring, reusable approach for structuring software; a blueprint for how a particular programming problem may be solved . 39, 40, 48, 55

DOM a term for the data structure used to represent components of a web page, namely the tree of HTML or XML tags and their associated properties. 48

ELN An electronic lab notebook (ELN) is a software tool for helping researchers to chronicle their day-to-day investigations and results by composing rich-text documents which consolidate data, code, plots, and natural-language research questions and analysis. 13

frontend the user-facing portion of a web app, e.g. the graphical interface displayed by a browser. 27

in-silico A designation applied to scientific endeavors which consist entirely of computer analysis of data, named in contrast with *in-vivo* biological experiments. 2, 14, 15

IoT The Internet of Things describes a near-future network infrastructure characterized by unprecedented device-to-device communication and ubiquitous Internet-capable sensors and actuators. 3, 18

JSON a simple text format, originally native to JavaScript, for encoding hierarchical data structures containing fields of many different types . 44

laboratory information management system (LIMS) A bundle of software tools for coordinating the activities of researchers, tracking inventory and data sets, and describing and monitoring experimental processes in one or more laboratories. 2

lazy-load a strategy where an application waits to load subcomponents until they are about to be used, decreasing the program's startup time and allowing it to depend on resources which may not be locatable until runtime information is available . 42, 48

microservice a small, single-purpose web application intended to communicate with a collection of other microservices. 27, 66

protocol stack a series of translation steps converting one communication protocol into another. 31

RDF a formalism for encoding graphs of semantic connections between entities via subject-verb-object triples which serves as the base language level for the W3C's semantic web standards. 20

research object a proposed type of rich electronic publication format for packaging data, executable procedures, and documentation in a single semantically-linked archive. 20

REST a software architecture where clients interact with servers by navigating a sequence of states or resources, each associated with a particular URI. 44

REST API An API adhering to Representational State Transfer (REST) principles. REST APIs are endpoints for issuing control and data commands over an HTTP interface, allowing web servers to expose functionality over the internet in a client-agnostic fashion. 44, 45, 48, 52, 64

RPC a programming abstraction where a sequence of network transactions is thought of as one machine remotely invoking a subroutine over the network, receiving its return value as a response. 40

scientific priority credit for being the first to publish or describe an invention or discovery. 12

semantic web an approach to knowledge management where Internet resources are annotated with groups of hyperlinks describing their relationships to other resources. 66

SQL a standardized language for accessing and managing databases using sets of search criteria. 50

switchboard a microservice responsible for determining which other microservices are active and making them available at appropriate URIs . 27–29

thin client a hardware or software component which acts as a lightweight portal connecting users to server-side functionality, involving little or no client-side software to use . 47

UI the portion of a software application concerned with accepting input from the user and producing output; often synonymous with graphical user interface (GUI). 47

URI a text string uniquely identifying an Internet resource. 28

VCS A set of software features related to tracking file revisions and allowing authors to revert files to previous states. 10

VISA an industry standard specification for the communication interface that a scientific testing or measurement instrument should provide. 53

WAMP a high-level application protocol built on top of WebSockets for allowing heterogeneous services to communicate via remote procedure calls and publish/subscribe event streams. 45

WMS A software package for creating and composing directed graphs of process phases and/or dependencies.. 13, 17

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Alberto Accomazzi et al. *Aggregation and Linking of Observational Metadata in the ADS*. 2016. eprint: `arXiv:1601.07858`.

[2] Agilent Technologies. *OpenLAB Software Suite*. (Accessed on 05/06/2016). URL: `https://www.agilent.com/en-us/products/software-informatics/openlabsoftwaresuite`.

[3] Thomas Allweyer. *BPMN 2.0*. BoD, 2010. ISBN: 3839149851, 9783839149850.

[4] Flask authors Armin Ronacher. *Flask (A Python Microframework)*. `http://flask.pocoo.org/`. (Accessed on 06/01/2016).

[5] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". In: *IEEE Softw.* 33.3 (May 2016), pp. 42–52. DOI: `10.1109/ms.2016.64`. URL: `http://dx.doi.org/10.1109/MS.2016.64`.

[6] Sean Bechhofer et al. "Research objects: Towards exchange and reuse of digital knowledge". In: (2010).

[7] Andrew Begel and Nachiappan Nagappan. "Usage and perceptions of agile software development in an industrial context: An exploratory study". In: *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. IEEE. 2007, pp. 255–264.

[8] C. Glenn Begley and Lee M. Ellis. "Drug development: Raise standards for preclinical cancer research". In: *Nature* 483 (2012). DOI: `10.1038/453531a`.

[9] Khalid Belhajjame et al. "Using a suite of ontologies for preserving workflow-centric research objects". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 32 (2015), pp. 16–42.

[10] Christian Bizer, Tom Heath, and Tim Berners-Lee. *Linked data-the story so far*. Ed. by Amit Sheth. Hershey, PA, 2011.

[11] Mikio L. Braun and Cheng Soon Ong. "Open Science in Machine Learning". In: *Implementing Reproducible Research*. Ed. by Victoria Stodden, Friedrich Leisch, and Roger D. Peng. CRC Press.

[12] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. “Data provenance: Some basic issues”. In: *FST TCS 2000: Foundations of software technology and theoretical computer science*. Springer, 2000, pp. 87–93.

[13] Jorge Cardoso, Robert P Bostrom, and Amit Sheth. “Workflow management systems and ERP systems: Differences, commonalities, and applications”. In: *Information Technology and Management* 5.3-4 (2004), pp. 319–338.

[14] Scott Chacon. *Pro Git*. 1st. Berkely, CA, USA: Apress, 2009. ISBN: 1430218339, 9781430218333.

[15] James Cheney et al. “Provenance: A Future History”. In: *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’09. Orlando, Florida, USA: ACM, 2009, pp. 957–964. ISBN: 978-1-60558-768-4. DOI: `10.1145/1639950.1640064`. URL: `http://doi.acm.org/10.1145/1639950.1640064`.

[16] Fernando Chirigati, Dennis Shasha, and Juliana Freire. “ReproZip: Using Provenance to Support Computational Reproducibility”. In: *Proceedings of the 5th USENIX Conference on Theory and Practice of Provenance*. TaPP’13. Lombard, IL: USENIX Association, 2013, pp. 1–1. URL: `http://dl.acm.org/citation.cfm?id=2482613.2482614`.

[17] The Open Science Collaboration. “Estimating the reproducibility of psychological science”. In: *Science* 349.6251 (2015). DOI: `10.1126/science.aac4716`. URL: `http://science.sciencemag.org/content/349/6251/aac4716`.

[18] Oscar Corcho et al. “Workflow-centric research objects: First class citizens in scholarly discourse.” In: (2012).

[19] CoreLIMS. *Workflow Management in the Core LIMS*. https://corelims.com/workflowmanagement.htm. (Accessed on 05/05/2016).

[20] ExpressJS developers. *Express - Node.js web application framework*. http://expressjs.com/. (Accessed on 06/01/2016).

[21] Node.js Developers. *Node.js*. https://nodejs.org/. (Accessed on 06/01/2016).

[22] Ulrich Dirnagl and Ingo Przesdzing. “A pocket guide to electronic laboratory notebooks in the academic life sciences”. In: *F1000Research* (Jan. 2016). DOI: `10.12688/f1000research.7628.1`. URL: `http://dx.doi.org/10.12688/f1000research.7628.1`.

[23] C ELLIOTT et al. “National Instruments LabVIEW: A Programming Environment for Laboratory Automation and Measurement”. In: *Journal of the Association for Laboratory*

Automation 12.1 (Feb. 2007), pp. 17–24. DOI: `10.1016/j.jala.2006.07.012`. URL: `http://dx.doi.org/10.1016/j.jala.2006.07.012`.

[24] Daniele Fanelli. “How Many Scientists Fabricate and Falsify Research? A Systematic Review and Meta-Analysis of Survey Data”. In: *PLoS ONE* 4.5 (May 2009). Ed. by Tom Tregenza, e5738. DOI: `10.1371/journal.pone.0005738`. URL: `http://dx.doi.org/10.1371/journal.pone.0005738`.

[25] Cat Ferguson, Adam Marcus, and Ivan Oransky. “Publishing: The peer-review scam”. In: *Nature* 515.7528 (Nov. 2014), pp. 480–482. DOI: `10.1038/515480a`. URL: `http://dx.doi.org/10.1038/515480a`.

[26] United States Food and Drug Administration. *Guidance for Industry Part 11, Electronic Records; Electronic Signatures - Scope and Application*. URL: `http://www.fda.gov/RegulatoryInformation/Guidances/ucm125067.htm`.

[27] IVI Foundation. *IVI Specifications*. http://www.ivifoundation.org/specifications/default.aspx. (Accessed on 05/23/2016).

[28] J. Freire and C. T. Silva. “Making Computations and Publications Reproducible with VisTrails”. In: *Computing in Science Engineering* 14.4 (July 2012), pp. 18–25. ISSN: 1521-9615. DOI: `10.1109/MCSE.2012.76`.

[29] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[30] Inc. GitHub. *GitHub*. https://github.com/. (Accessed on 06/01/2016).

[31] *GNU Emacs*. https://www.gnu.org/software/emacs/. (Accessed on 05/07/2016).

[32] AngularJS developers Google. *One framework. - Angular 2*. https://angular.io/. (Accessed on 06/01/2016).

[33] The HDF Group. *Hierarchical Data Format, version 5*. http://www.hdfgroup.org/HDF5/. 1997-NNNN.

[34] Megan L. Head et al. “The Extent and Consequences of P-Hacking in Science”. In: *PLOS Biology* 13.3 (Mar. 2015), e1002106. DOI: `10.1371/journal.pbio.1002106`. URL: `http://dx.doi.org/10.1371/journal.pbio.1002106`.

[35] Fuguo Huang. “Web Technologies for the Internet of Things”.

[36] John P. A. Ioannidis. “Why most published research findings are false”. In: *PLoS Med* (2005). URL: `http://dx.doi.org/10.1371/journal.pmed.0020124`.

[37] Donald E. Knuth. “Literate Programming”. In: *Comput. J.* 27.2 (May 1984), pp. 97–111. ISSN: 0010-4620. DOI: `10.1093/comjnl/27.2.97`. URL: `http://dx.doi.org/10.1093/comjnl/27.2.97`.

[38] Neal Leavitt. “Will NoSQL Databases Live Up to Their Promise?” In: *Computer* 43.2 (Feb. 2010), pp. 12–14. ISSN: 0018-9162. DOI: `10.1109/MC.2010.58`. URL: `http://dx.doi.org/10.1109/MC.2010.58`.

[39] Albert L. Lederer et al. “The technology acceptance model and the World Wide Web”. In: *Decision Support Systems* 29.3 (Oct. 2000), pp. 269–282. DOI: `10.1016/s0167-9236(00)00076-2`. URL: `http://dx.doi.org/10.1016/S0167-9236(00)00076-2`.

[40] Jens Lehmann et al. “DBpedia–a large-scale, multilingual knowledge base extracted from Wikipedia”. In: *Semantic Web* 6.2 (2015), pp. 167–195.

[41] James Lewis and Martin Fowler. *Microservices*. http://martinfowler.com/articles/microservices.html. (Accessed on 05/09/2016). Mar. 2014.

[42] H. Li et al. “Low Power Multimode Electrochemical Gas Sensor Array System for Wearable Health and Safety Monitoring”. In: *IEEE Sensors Journal* 14.10 (Oct. 2014), pp. 3391–3399. ISSN: 1530-437X. DOI: `10.1109/JSEN.2014.2332278`.

[43] LIMSWiki. *LIMS vendor*. http://www.limswiki.org/index.php/LIMS_vendor. (Accessed on 05/06/2016).

[44] S. Marco and A. Gutierrez-Galvez. “Signal and Data Processing for Machine Olfaction and Chemical Sensing: A Review”. In: *IEEE Sensors J.* 12.11 (Nov. 2012), pp. 3189–3214. DOI: `10.1109/jsen.2012.2192920`. URL: `http://dx.doi.org/10.1109/JSEN.2012.2192920`.

[45] Catherine C. Marshall and Frank M. Shipman. “Which Semantic Web?” In: *Proceedings of the Fourteenth ACM Conference on Hypertext and Hypermedia*. HYPERTEXT ’03. Nottingham, UK: ACM, 2003, pp. 57–66. ISBN: 1-58113-704-4. DOI: `10.1145/900051.900063`. URL: `http://doi.acm.org/10.1145/900051.900063`.

[46] *MATLAB R2016a*. Natick, Massachusetts, 2016.

[47] James McCartney. “Rethinking the computer music language: SuperCollider”. In: *Computer Music Journal* 26.4 (2002), pp. 61–68.

[48] Wes McKinney. “Data Structures for Statistical Computing in Python”. In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 51–56.

[49] Microsoft Inc. *Microsoft Word*. 2013. URL: `https://products.office.com/en-us/word`.

[50] Paolo Missier, Khalid Belhajjame, and James Cheney. “The W3C PROV Family of Specifications for Modelling Provenance Metadata”. In: *Proceedings of the 16th International Conference on Extending Database Technology*. EDBT ’13. Genoa, Italy: ACM, 2013, pp. 773–776. ISBN: 978-1-4503-1597-5. DOI: `10.1145/2452376.2452478`. URL: `http://doi.acm.org/10.1145/2452376.2452478`.

[51] C. Mohan. “History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla”. In: *Proceedings of the 16th International Conference on Extending Database Technology*. EDBT ’13. Genoa, Italy: ACM, 2013, pp. 11–16. ISBN: 978-1-4503-1597-5. DOI: `10.1145/2452376.2452378`. URL: `http://doi.acm.org/10.1145/2452376.2452378`.

[52] Lorraine Morgan and Patrick Finnegan. “Benefits and drawbacks of open source software: an exploratory study of secondary software firms”. In: *Open Source Development, Adoption and Innovation*. Springer, 2007, pp. 307–312.

[53] Tom Oinn et al. “Taverna: lessons in creating a workflow environment for the life sciences: Research Articles”. In: *Concurrency and Computation: Practice & Experience* 18 (June 2006). ISSN: 1532-0626. DOI: `10.1002/cpe.v18:10`. URL: `http://dl.acm.org/citation.cfm?id=1148437.1148448`.

[54] W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. Available at `http://www.w3.org/TR/owl2-overview/`. W3C Recommendation, 27 October 2009.

[55] Fernando Pérez and Brian E. Granger. “IPython: a System for Interactive Scientific Computing”. In: *Computing in Science and Engineering* 9.3 (May 2007), pp. 21–29. ISSN: 1521-9615. DOI: `10.1109/MCSE.2007.53`. URL: `http://ipython.org`.

[56] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. “Semantics and Complexity of SPARQL”. In: *ACM Trans. Database Syst.* 34.3 (Sept. 2009), 16:1–16:45. ISSN: 0362-5915. DOI: `10.1145/1567274.1567278`. URL: `http://doi.acm.org/10.1145/1567274.1567278`.

[57] Michael Pilato. *Version Control With Subversion*. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2004. ISBN: 0596004486.

[58] R Development Core Team. *R: A Language and Environment for Statistical Computing*. ISBN 3-900051-07-0. R Foundation for Statistical Computing. Vienna, Austria, 2008. URL: `http://www.R-project.org`.

[59] Guillermo Rauch. *Socket.IO*. http://socket.io/. (Accessed on 06/01/2016).

[60] Guido Rossum. *Python Reference Manual*. Tech. rep. Amsterdam, The Netherlands, The Netherlands, 1995.

[61] Michael Rubacha, Anil K. Rattan, and Stephen C. Hosselet. "A Review of Electronic Laboratory Notebooks Available in the Market Today". In: *Journal of Laboratory Automation* 16.1 (Feb. 2011), pp. 90–98. DOI: `10.1016/j.jala.2009.01.002`. URL: `http://dx.doi.org/10.1016/j.jala.2009.01.002`.

[62] *Simulink*. Natick, Massachusetts. URL: `http://www.mathworks.com/help/simulink/`.

[63] Pivotal Software. *RabbitMQ - Messaging that just works*. https://www.rabbitmq.com/. (Accessed on 06/01/2016).

[64] David I Spivak and Robert E Kent. "Ologs: a categorical framework for knowledge representation". In: *PLoS One* 7.1 (2012), e24274.

[65] Rod Stephens. *Beginning Software Engineering*. 1st. Birmingham, UK, UK: Wrox Press Ltd., 2015. ISBN: 1118969146, 9781118969144.

[66] StrongLoop. *LoopBack Framework*. https://strongloop.com/node-js/loopback-framework/. (Accessed on 06/01/2016).

[67] Tavendo. *Crossbar.io*. http://crossbar.io/. (Accessed on 06/01/2016).

[68] Sabu M. Thampi. "Introduction to Distributed Systems". In: *CoRR* abs/0911.4395 (2009). URL: `http://arxiv.org/abs/0911.4395`.

[69] Stefán van der Walt, S Chris Colbert, and Gaeël Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Comput. Sci. Eng.* 13.2 (Mar. 2011), pp. 22–30. DOI: `10.1109/mcse.2011.37`. URL: `http://dx.doi.org/10.1109/MCSE.2011.37`.

[70] Huaiqing Wang and Chen Wang. "Open source software adoption: A status report". In: *Software, IEEE* 18.2 (2001), pp. 90–95.

[71] Zhe Wang et al. "Highly Sensitive Capacitive Gas Sensing at Ionic Liquid–Electrode Interfaces". In: *Analytical Chemistry* 88.3 (Feb. 2016), pp. 1959–1964. DOI:

10.1021/acs.analchem.5b04677. URL: http://dx.doi.org/10.1021/acs.analchem.5b04677.

[72] Zhe Wang et al. “Methane–oxygen electrochemical coupling in an ionic liquid: a robust sensor for simultaneous quantification”. In: *The Analyst* 139.20 (June 2014), pp. 5140–5147. DOI: 10.1039/c4an00839a. URL: http://dx.doi.org/10.1039/C4AN00839A.

[73] Matthew West. “Complex Systems in Knowledge-based Environments: Theory, Models and Applications”. In: Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. Chap. Ontology Meets Business - Applying Ontology to the Development of Business Information Systems, pp. 229–260. ISBN: 978-3-540-88075-2. DOI: 10.1007/978-3-540-88075-2_9. URL: http://dx.doi.org/10.1007/978-3-540-88075-2_9.

[74] E. Wolff. *Microservices: Flexible Software Architectures*. CreateSpace Independent Publishing Platform, 2016. ISBN: 9781523361250. URL: https://books.google.com/books?id=X7YzjwEACAAJ.

[75] Wolfram Research, Inc. *Mathematica 8.0*. Version 0.8. 2010. URL: https://www.wolfram.com.

[76] Katy Wolstencroft. *myExperiment - Workflows - Blast_Align_and_Tree (Katy Wolstencroft) [Taverna 2 Workflow]*. http://www.myexperiment.org/workflows/3369.html. (Accessed on 05/08/2016). Jan. 2013.

[77] Zetta. *Zetta - An API-First Internet of Things (IoT) Platform - Free and Open Source Software*. http://www.zettajs.org/. (Accessed on 05/04/2016).