POOLSIDE: A COMPUTATIONAL NOTEBOOK

BY
DAVID WYDE

A Thesis

Submitted to the Division of Natural Sciences
New College of Florida
in partial fulfillment of the requirements for the degree
Bachelor of Arts in Computer Science
Under the sponsorship of Chris Hart

Sarasota, Florida
May, 2011

Table of Contents

List of Figures iii

List of Tables iii

Abstract iv

Introduction 1
- Motivation . . . 1
- Existing Solutions . . . 3
- Summary . . . 10

Background and Implementation 12
- Architecture . . . 12
- A Computational Notebook . . . 13
- Web Applications . . . 14
- CouchDB . . . 18
- Code Evaluation Server . . . 29

Results 37
- Future Work . . . 37

Appendix 39
- Past Architectures . . . 39

Bibliography 42

List of Figures

1 Architectural Diagram 13
2 Poolside's Notebook Interface 14

List of Tables

1 Applications vs. Features 11

Abstract

Data analysis is a central aspect of modern computational science. The scientific method emphasizes reproducibility of results, but intermediate steps in analytical processes are not always recorded and published. To verify or extend an existing experiment requires knowledge of each step in the original analysis. For data-centric disciplines like genomics, a complete “analysis pipeline” is the computational equivalent of a formal lab protocol. We have created a software project, called *Poolside*, that aims to help scientists programmatically analyze data in a reproducible way.

Poolside is an open source web application with tools for computation and visualization. All of Poolside’s features are accessible from a web browser. Its users don’t need to download additional software. Poolside allows users to write and run code in the Python and Ruby programming languages. Poolside stores data persistently, and also has features for data visualization. It is intended to act as a “computational notebook”. Poolside is still in development; however, it already has some useful capabilities, and it provides a platform with which more sophisticated tools can be built.

Introduction

Motivation

Modern computational tools must cope with massive amounts of data. Advances in technology across many fields, including genomics, finance, and astronomy, have caused data to be generated more quickly than people can analyze it [21]. Today's computational scientists look for trends buried in large piles of raw data. In this age of "big data", good analytical software is extremely important. Geographically distributed groups of collaborators need a way to deal with large data sets, in both business and academia.

Reproducible results are a key component of the scientific method [36], and computational methods are an increasingly important part of modern science. To independently verify an experiment requires repeating each step in the original data analysis. This "analysis pipeline" serves as an experimental protocol; simply releasing graphs and raw data does not provide a way to reproduce computational results. There are many tools for domain-specific analyses, but much software fails to keep track of individual computations.

Our Solution

We are developing software to make reproducible analysis more convenient, and to enable collaborative extension and computational experimentation. We designed our project to be a hybrid of note-taking and data analysis software: an electronic scientific lab notebook. It provides streamlined and pluggable computational engines that can be used to analyze data and notes. This computational notebook and analysis platform, called *Poolside*, allows users to securely share their analyses on the internet.

Poolside is an interactive *web application*. It requires minimal setup by its users: all of Poolside's features are accessible from a web browser. This facilitates collaboration, because anyone with an internet connection can view a project. There is no need to download any extra software.

The main idea behind Poolside's *computational tools* is to store blocks of input code with their associated outputs. This is a key part of reproducible analysis: keeping track of the code that generates each data set. Plain text notes can act as comments, describing the analysis.

Visualizations are an important tool for understanding data. It can be hard to make sense of a table with millions of rows, but a simple bar chart is often comprehensible. Poolside contains the foundations for data visualization features, but the current version provides only a single type of graph.

Poolside is *redistributable*. We are releasing it under a free software license, likely the GNU GPL[1]. The end vision of the project is that it could scale to be useful to individual users as well as larger organizations or agencies.

Summary

We have designed Poolside to be a "computational notebook". Such an application should enable its users to perform several fundamental tasks: record and analyze data, display graphs and other visualizations, share (publish) results, and extend computational analysis. It should also require minimal setup by its end users.

Poolside provides users access to a complete computational environment and data exchange platform. It is constructed on top of several open source technologies and software projects, and can be accessed from a single web browser interface. At the most

[1] General Public License -- a canonical "copyleft" license.

basic level, it records plain text notes. Poolside also has computational tools and data visualization capabilities. It shares individual features with a large number of existing applications, but we believe that Poolside fills a useful niche in the software ecosystem.

Existing Solutions

This section is a survey of existing software projects that share some of our goals. It is not intended to be complete, but instead serves as an overview. This discussion will frame the rest of the thesis, providing context for some of our design choices. The applications are organized into categories, somewhat artificially. The section concludes with a table that compares all of the software.

Note-taking Applications

As a starting point for this discussion, we will examine applications that are primarily designed to record notes. This type of software achieves one of Poolside's primary design requirements, despite differing use cases.

Remember the Milk is a simple "task management" application. Other online to-do list software, such as Google's *Gmail Tasks*, typically has similar features. These tools provide a quick and easy way to record notes, but they are limited in functionality.

Evernote is a data organization application with many features. It allows users to store arbitrary collections of data, a mix of text and media. Evernote is web-enabled by design: it syncs data between different client devices, and its "Web Clipper" feature saves content directly from web pages. Evernote's ability to save everything in one place is a nice feature, although the free version restricts the type of files that can be attached to notes. Evernote also enables users to share notes, although it lacks computational tools.

Poolside's underlying software, like Evernote, has the ability to associate images with text notes. This is a feature that we might like to implement in the future, to save rendered plots alongside the source of their data.

Wikis

Informally, a wiki is a set of documents that are edited in a collaborative way, with an emphasis on referencing other documents in the wiki. Two of the three projects mentioned in this subsection are not conventional wikis. They are designed for personal use, rather than collaboration. Wikis often permit third-party extensions, to add new features.

Tomboy bridges the boundary between note-taking software and wikis: its main conceptual structure is a "note", and it permits internal links. These cross-note references allow Tomboy to function as a wiki. It works on a variety of desktop platforms, and an online service for Tomboy is in early development [13].

MediaWiki is an open source wiki software project. It helps online communities share static content; *Wikipedia* is built with MediaWiki. MediaWiki can gain additional features via extensions. One plugin allows users to find and visualize data using the *SPARQL* query language [34], and another enables plotting with the *GraphViz* library [33].

TiddlyWiki takes a creative approach to wiki software, storing an entire wiki in a single HTML file. Another of TiddlyWiki's main features is that it's a purely client-side technology: its base requirements are a modern web browser and an ability to write to the wiki file. It's somewhat limited to personal use by default, but there are some extensions for sharing [37].

Tomboy and TiddlyWiki are more for personal use than online collaboration, although

each has some intriguing features (especially TiddlyWiki). MediaWiki represents wikis as an interesting way to share information, and has good visualization plugins. Wikis generally seem not to focus on computation, but it might be possible to overcome this limitation with extensions.

TiddlyWiki is a good example of freely redistributable software with minimal dependencies. We want Poolside to include embedded visualization tools, like MediaWiki (with extensions).

Real-time Collaboration Software

The applications discussed in this section allows users to simultaneously edit the same document. Real-time collaborative tools have several uses in software development, especially with team projects.

Microsoft OneNote is part of the proprietary *Microsoft Office* suite. Like Evernote, OneNote can store notes of mixed text and media. OneNote is designed both for the desktop and to allow live online collaboration on notebooks [35].

Gobby is a collaborative text editor. It's a desktop application that communicates via a protocol called *Infinote* [31]. An infinote server connects clients to "sessions", in which users can edit documents and chat in real time. Gobby is free software, and infinote is an open standard.

Google Wave was a real-time messaging service hosted by Google. Its features were an amalgamation of instant messaging, email, collaborative document editing, and several other services. One of Wave's most interesting features was its blurring of the distinction between email, chat, and wiki-style editing. Google stopped developing Wave in August 2010, but the project was open sourced and relocated to the Apache Software Foundation [26].

Google Docs is an online office suite with features for collaboration. Its word processor allows users to simultaneously edit a document, and displays each user's activity in real time. This feature, combined with instant messaging, makes Google Docs an excellent tool for collaborative writing.

The projects mentioned in this section have some compelling features. While they lack any sort of computational tools, they approach the problem of distributed collaboration in a novel way. Poolside is unlikely to enable live shared editing in the near future; however, that would greatly enhance its collaborative capabilities and could potentially make use of the open Wave protocol [4]. Microsoft OneNote's ability to work both online and offline is a desirable feature for Poolside.

Spreadsheets

Spreadsheet software is useful for storing tabular data. In addition to application-specific file formats, *.csv* files are supported by nearly all spreadsheet applications. This generally makes it possible to share raw data across platforms. Additionally, many spreadsheet programs support embedded scripting via "macros".

Microsoft Excel, part of Microsoft Office, is a popular spreadsheet application. It allows macros in *Visual Basic for Applications* (VBA). There are open source alternatives to Excel, such as *OpenOffice Calc* and *Gnumeric*. These desktop spreadsheet programs are well-established, and all three can generate charts from data.

Google Docs Spreadsheet and *Zoho Sheet* are part of online office suites. Each is a full-featured spreadsheet program that allows realtime collaboration. Zoho Sheet allows VBA macros like Microsoft Excel, and provides users with programmatic access to their data [19].

There is an open source project called *pyspread*, a desktop program still in beta [32].

It uses the *Python* programming language for computation. From our perspective, this is an important feature. Rather than providing users with a restricted macro language, pyspread enables arbitrary analysis. Running scripts written by unknown users becomes a security issue, but pyspread could prove useful for personal analyses.

Zoho Sheet provides a very solid-looking online spreadsheet, and its data accessibility might help compensate for its proprietary nature. A programmatic interface to data is an important feature for many web applications. We don't want to limit our application exclusively to spreadsheets or relational[2] data, but pyspread's use of Python is noteworthy.

Data Analysis Tools

The projects in this subsection are designed to process and plot data. They provide convenient wrappers around programming languages, especially Python. These tools focus to some extent on scientific computing, but they have many general-purpose applications.

IPython is an interactive Python shell. Python already ships with a simple interactive shell, but IPython has several features that Python's default interpreter lacks: colorful output and persistent command history, for example. The development version includes features for parallel computing, and uses a message passing system called *ZeroMQ* that is particularly useful for networked applications [3]. An initial implementation of a simple IPython web client was announced in October 2010, but it hasn't received further development [25].

Galaxy unifies a variety of data analysis tools into a single platform. It's designed for analysis of genomic data, targeting the sharing of computational tools and analyses. Galaxy tracks each step in an analysis "workflow". These data pipelines allow the

[2] We will define "relational" in the section on databases.

output of one command to act as an input to one or more further steps. Galaxy provides a limited set of predefined scripts for data manipulation. This abstraction means that certain operations are very convenient (especially for non-programmers), but other analyses are impossible.

Sage is an open source mathematics package. It provides a single interface to many different computational tools, and aims to be a free alternative to proprietary math software such as *MATLAB* and *Mathematica* [11]. Sage wraps almost 100 independent projects into one package, mostly mathematical and scientific software. Sage also comes with *SageNB*: an online "notebook" interface to Sage's wide array of tools, both computational and graphical [10]. It avoids external dependencies by including everything in a single download.

We incorporated both Sage and IPython into earlier versions of our project, but eventually decided that having few installation dependencies is a bigger priority than using either one's architecture as a foundation. Galaxy is also a very solid platform on which to build, and we utilized it in an earlier design of our project. Galaxy's domain-specific nature limits its usefulness as a general computational tool, but it's a good choice for genomic applications.

Cloud-based Solutions

"Cloud computing" is a way to store data and provide services online. Users indirectly access resources "in the cloud", i.e., from a pool of remote machines that are owned by some external service provider. A cloud-based web application receives queries, performs the actual computations on other machines, and then returns the results to each client.

PiCloud aims to make servers transparent to Python programmers. Rather than execute programs on their own machines, users wrap each function call via a provided Python

library. PiCloud executes this code “in the cloud”, and charges users based on running time. It also provides a simple interface to running Python functions in parallel.

Dirigible is a browser-based spreadsheet with each cell computed from a Python expression and represented as a Python object. It has a lot in common with pyspread, but focuses on cloud computing. It also features programmatic access to the contents of sheets. Dirigible is currently in beta and provides a free trial, but it generally charges by compute time.

Both PiCloud and Dirigible do a good job of abstracting details for their users. In typical cloud computing fashion, users can submit a request, wait while their code runs “in the cloud”, and get a result. It doesn’t matter if the servers are geographically distributed, if they’re running on the same hardware, etc. We want Poolside to work in a somewhat similar fashion, in which users don’t need to worry about anything other than performing their analysis.

Online Text Editors and Integrated Development Environments (IDEs)

This last group of software includes tools to write code in a web browser. There are quite a few projects in this realm, and more on the way. For example, the popular *Eclipse* desktop IDE is building an online version of its software [24].

JsFiddle is a tool for web development. Users code in the standard front end languages HTML, CSS, and JavaScript, and view the resulting web page. Each project has its own URI[3], which allows users to share code easily. All code runs in the user’s browser, simplifying some aspects of managing the application (especially with regard to security).

Ajax.org Cloud9 Editor (Ace) is an online code editor. It supports syntax highlighting

[3] Uniform Resource Identifier: a specific location on the internet.

and on-the-fly error checking for a number of languages. Ace does not execute code by itself, but a command line feature is available via an associated project called *Cockpit* [5].

Codepad is a web application that allows users to test and share code. It can execute programs in a variety of languages, both compiled and interpreted. Like jsFiddle, codepad assigns a URI to each "paste". Codepad has strict security features, to protect the main server from malicious code [29]. This software fulfills many of our project's goals; however, it does not appear to be open source (although it is free to use).

Poolside's user interface is not particularly polished. It might make sense to look at how jsFiddle and Ace implement certain features, e.g., on-the-fly syntax highlighting in a web browser. Codepad's security features are more rigorous than Poolside's, which is very important when executing arbitrary user code from the internet. Future versions of Poolside should include stronger security measures.

Summary

Poolside's architecture draws on ideas from a variety of other projects. Despite its apparent similarities to existing software, we believe that Poolside has the potential to be useful as a tool for analyzing data in a reproducible way, while also allowing users to share, extend, and explore computational analyses more readily. Our current implementation is certainly not complete, but it provides a foundation on which to build.

The following table is one way to compare the different software discussed above. The number of check marks should not be taken as a "score". Microsoft Excel has three of our four desired features, compared to codepad's two; however, the latter is much more similar to our project.

Table 1: Applications vs. Features

	Free Software	Computation	Visual Data	Online
Ajax.org Cloud9 Editor	X	X		X
codepad		X		X
Dirigible		X	X	X
Evernote			X	X
Galaxy	X	X	X	X
Gobby	X			X
Google/Apache Wave	X		X	X
Google Docs			X	X
IPython	X	X	X	
jsFiddle	X	X		X
MediaWiki	X		X	X
Microsoft Excel		X	X	X
Microsoft OneNote			X	X
PiCloud		X		X
pyspread	X	X	X	
Remember the Milk				X
Sage	X	X	X	X
TiddlyWiki	X		X	X
Tomboy	X			X
Zoho Sheet		X	X	X

Background and Implementation

In the previous chapter we gave a brief overview of our software, and several projects with similar features. We provided a general description of Poolside: an open source web application with tools for computation and visualization. We will now explain the technical details of our implementation, and discuss our design decisions.

Architecture

To build Poolside with the features that we have described, we must construct several different components. In order to be a web application, Poolside needs a web browser user interface. This requires an associated web server, to which clients will connect. It needs a computational back end tool, which will execute users' code. Poolside must have a way to store data persistently, if it is to be useful as a notebook. Finally, we would like Poolside to have an authentication mechanism, to prevent vandalism. Rather than construct all of these pieces from scratch, we built Poolside on top of free and open source software projects.

The image below depicts the relationships between Poolside's architectural components. In the diagram, the *client* is a web browser. Users interact with Poolside as a web application, via a "notebook" interface. *HTTP* (Hypertext Transfer Protocol) is the standard means by which different entities communicate online. *HTTPS* (HTTP Secure) is a way to protect an HTTP connection with encryption.

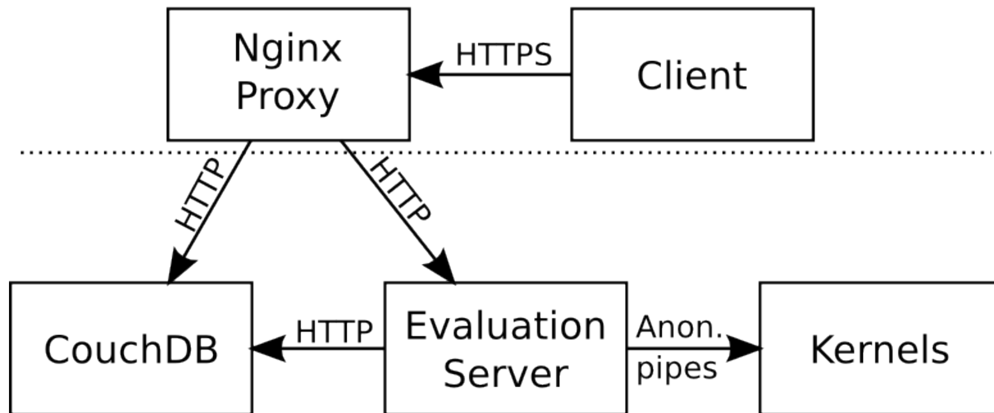

Figure 1: Architectural Diagram

The *nginx* proxy is a sort of "middleman" between a client and various server-side components. It provides a single entry point to the rest of the application, and enables HTTPS between clients and Poolside. *CouchDB* is a scalable database, a web server for the notebook interface, and an authentication system. The *evaluation server* handles computational requests. It manages the *kernels* that actually run users' code. The evaluation server communicates with kernels via an operating system feature called *anonymous pipes*.

This was intended to be a quick tour of Poolside's implementation, a sort of blueprint. After briefly discussing Poolside from an end user's perspective, we will examine the individual architectural pieces more thoroughly.

A Computational Notebook

Poolside's primary user interface is a "notebook", accessible in any web browser. The basic conceptual entity of Poolside is a *cell*: an individual computation or note. Cells can be plain text, or they can execute code in a programming language that Poolside

supports. Cells are grouped together into collections called *worksheets*. An example of a Poolside worksheet is below:

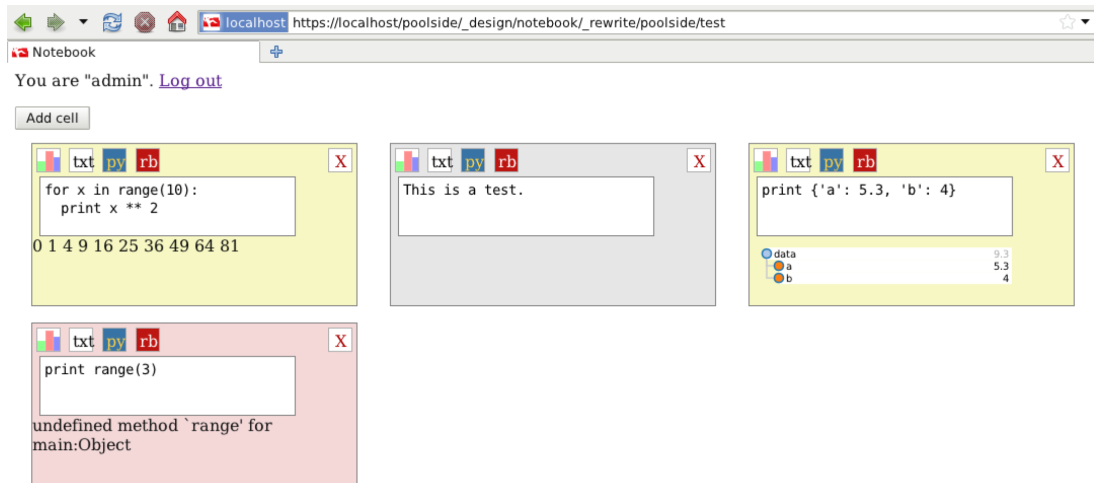

Figure 2: Poolside's Notebook Interface

The interface is fairly plain, but it provides access to all of Poolside's features. Each box is a cell. Gray cells correspond to plain text. Yellow and pink cells correspond to computational input/output pairs: code written in the *Python* and *Ruby* programming languages, respectively. The leftmost button in each cell is for client-side data visualization. It converts output text into a graphic, if possible. The rightmost cell demonstrates this feature. The upper-right button permanently deletes a cell. An important feature of Poolside is that users can type code into a cell's input, click a button ("py" for Python, "rb" for Ruby), and receive the resulting output. Poolside is a web browser interface to programming language environments.

Web Applications

Poolside is a *web application*. In contrast to a *desktop application*, it accesses content over a TCP/IP[4] network (e.g., the internet), via a web browser. The line between these two categories is often blurry, but Poolside is fundamentally web-centric. It fea-

tures a clear conceptual split between client and server. Poolside's notebook frontend is designed for end users; the CouchDB and evaluation servers provide services for the application, but users will generally not interact with them directly.

We describe web applications to help explain the context in which we have developed Poolside. Several of its features reflect themes common to most web-oriented software.

Web Standards

Open web standards are an important concept in the development of web applications. Standards specify technical details of underlying technologies such as HTML (HyperText Markup Language), HTTP (Hypertext Transfer Protocol), and Unicode. Groups like the World Wide Web Consortium (W3C) and Internet Engineering Task Force (IETF) coordinate the development of web standards.

Standards can help maintain consistency across platforms. Certain browsers might render a web page incorrectly; however, they all generally follow the HTML 4.01 specification (and increasingly HTML5). Poolside uses HTML 4, Cascading Style Sheets (CSS2), and JavaScript (an implementation of the ECMAScript standard) for its notebook interface. It also relies on Scalable Vector Graphics (SVG), an XML-based W3C standard, for data visualization. Standards such as these permeate modern web application development.

HTTP is the foundational standard for the World Wide Web. It is the core of web services and applications, because it allows clients to send data to a server in a platform-neutral way. Every HTTP request must specify a *method*, such as GET, POST, or DELETE. This is the "verb" that describes a request. A GET request retrieves data from the server, but it won't modify anything; a POST indicates an intention to change data on the server. HTTP requests can send additional parameters to a server; a login re-

[4] Transmission Control Protocol is an underlying protocol on which the World Wide Web relies.

quest might typically include a username and password. Both CouchDB and Poolside's evaluation server communicate with clients via HTTP.

Front End Web Development

Front end web development generally consists of three languages: HTML, CSS, and JavaScript. HTML is a markup language: it defines the structure and content of a web page. CSS specifies the presentation of web pages: layout and colors, for example. Client-side JavaScript is used to make web sites dynamic, e.g., any custom response to user input.

AJAX (Asynchronous JavaScript and XML) is an important intersection of various web technologies. Dynamically loading new data from a server into a web page has, in the past, required complete page reloads, which is slow and does not provide a good user experience compared to desktop applications. AJAX is a way to avoid constantly refreshing pages in interactive web applications, providing a user experience that can rival the interactivity of desktop software. It allows a web page to send content to a server via JavaScript, continue functioning while the server processes its HTTP request, and respond when the data is ready. This technique allows web applications to be exceptionally interactive.

Like many web applications, Poolside's notebook interface relies heavily on AJAX. When a cell needs to be saved, Poolside makes an asynchronous HTTP request to CouchDB in the background (i.e., without user intervention). The notebook also sends requests to the evaluation server via AJAX. This allows users to continue working during long-running computations. AJAX helps to make Poolside responsive.

APIs

Web applications often provide application programming interfaces (APIs) for extensibility. An API is a set of operations that external developers can programmatically perform with an application. For example, an online photo gallery's API might allow users to add an album or upload pictures. APIs act as an alternative to a web application's primary user interface, and generally provide a controlled subset of available features. CouchDB has an HTTP API: that is the means by which users query the database. Poolside's evaluation server has a sort of implicit HTTP API, due to its simplicity. All of its features are accessible via HTTP requests. Even though the evaluation server does not define a special interface for third-party consumption, its users can evaluate code from any HTTP client.

Component Coupling

Poolside doesn't expect its end users to know anything about its implementation details. Following the "cloud computing" paradigm, Poolside allows CouchDB and the evaluation server to run on different machines. These two components communicate via HTTP, and each server simply requires the other's address in order to send messages.

This brief discussion of web applications leads into a more involved analysis of Poolside's implementation details.

CouchDB

Databases

Poolside needs somewhere to put its data, i.e., users' worksheets and cells. Software with potentially large amounts of data will often rely on *databases*. A database provides a convenient way to store and query data. We will discuss two categories of databases: *relational* and *NoSQL*.

Relational databases store records as "rows". Each row is a tuple, containing a value that corresponds to a list of columns. Columns are effectively headings on a "table" that contains the rows. Relational databases are good at storing tabular data, due to this structure of rows and columns.

In the relational data model, each "entity" in a data set is stored in its own table. Tables are related by one-to-one, one-to-many, or many-to-many relationships. Each relational database has a *schema*: a strict set of rules to limit the data that can be saved to the database. If a schema specifies that the `ID` column of a particular database must be an integer, then users will not be able to insert a row that has a string `"hello"` in that column's position.

Users interact with a relational database by SQL (Structured Query Language). Programmers use SQL to add rows, modify a constraint on a column, make a query, and so on. SQL is the characteristic interface to relational databases, and it's been standardized by the International Organization for Standardization (ISO) [22].

Among other things, a relational database management system (RDBMS) helps to maintain data integrity across the entirety of the database. Most implementations strive to prevent data anomalies. For example, *foreign key constraints* are a mechanism to prevent updates that would make data inconsistent. It is more difficult to maintain data

integrity in a collection of text files.

Some of the most well-known relational databases are Oracle, MySQL, PostgreSQL, and SQLite. There are a few liabilities associated with relational databases, and the relational model in general. Other data models are sometimes a better fit for particular data sets. A *graph database* might be more suitable for data with high interconnectivity, e.g., various networks [17]. Despite the extreme popularity of relational databases, there are alternative ways to store data.

A different style of database, dubbed "NoSQL", has recently been trendy in large-scale web applications. This class of databases is characterized by a lack of SQL. NoSQL databases are often structured as key-value stores, instead of a relational model. They can be schema-free, document-oriented (as opposed to the tabular relational data), or both. Examples of NoSQL databases include MongoDB, Redis, Tokyo Cabinet, and CouchDB. We chose to use CouchDB as the core of Poolside.

CouchDB

CouchDB is a document-oriented, schema-free, NoSQL database. It is geared toward web services, because its API is accessible via HTTP. It is written in the concurrency-friendly Erlang programming language, and uses JavaScript for most server-side configuration.

CouchDB stores each record as a JSON (JavaScript Object Notation) document. JSON is a data interchange format, based on a subset of JavaScript. Its basic data structures are key-value pairs, ordered lists, and values: numbers, strings, booleans, and "null" [6]. Key-value pairs and lists can be nested. Each CouchDB document has associated metadata: a unique identification string that's somewhat analogous to a primary key in relational databases, and a revision string for managing updates.

CouchDB documents are not required to conform to a predefined schema, in contrast to records in relational databases. CouchDB provides a mechanism to enforce constraints before a document can be updated ("validation functions"), but the default format is arbitrary JSON. Freedom from a rigid schema permits users to decide how to organize their data, helping to make Poolside useful across domains. In addition, JSON objects are often a natural representation of data types available in high-level dynamic programming languages. This allows analyses in Poolside to be extended with other computational tools.

Querying

CouchDB documents are accessible via standard HTTP methods such as GET, POST, PUT, and DELETE. An individual document can be accessed via a URI that corresponds to its unique ID. A CouchDB database can provide a set of predefined queries for users to perform. One example is map/reduce: a technique for transforming data, popular in NoSQL data stores. Map/reduce draws its name from the functional programming *map* and *reduce* concepts.

A map function applies the same function to every element in a list, returning a new list that contains the transformed elements. A reduce function repetitively applies a function to consecutive elements in a list, condensing the list to a single value. E.g., summing a list of numbers. A map function, possibly with an associated reduce function, is the basis of a CouchDB *view*. Views are the main way to retrieve multiple CouchDB documents in one HTTP request. CouchDB indexes its views in a B+tree data structure, a variation of the B-tree. This allows for fast retrieval of documents from disk [16].

CouchDB's *show* and *list functions* transform JSON documents into a different format. Show functions operate on individual documents; list functions are applied to the results of a view. A show function, for example, might output a CouchDB document as XML

or HTML.

Poolside's basic CouchDB views do not involve reduce operations, but a future version might utilize them to provide users with efficient access to predefined queries. The map function to get cell documents from a worksheet document relies on a CouchDB feature called "linked documents" [39]. It transforms a worksheet's list of cell IDs into the corresponding cell documents (in order). Each element in the view's results is a child cell of the specified worksheet.

To render the notebook, Poolside runs these results through a CouchDB list function. Poolside uses a list function to render a collection of cells as HTML. Each (JSON) document is transformed into an HTML template: a cell "widget". The underlying structure of all HTML cells is the same, but each one displays the distinct content (e.g., input and output) of its original document.

Replication

CouchDB features *bidirectional replication*. This is useful if two copies of the same database get out of sync, e.g., one of them goes offline. A successful replication will synchronize the document updates and deletions from one database into another. Each replication request must have a `source` parameter and a `target` parameter.

Replication could allow Poolside to have a "work offline" mode. Clients would synchronize with the main CouchDB instance to obtain a working copy of the application. Then, after making offline changes, they could push their updated documents back into the main CouchDB database. This is not implemented yet, but it should be possible.

Roles

In addition to being a database, CouchDB also acts as a web server, and it provides a built-in authentication system. We rely heavily on both of these features.

Authentication

Poolside needs some sort of authentication system; otherwise, anyone can modify any notebook or kernel. We would like each person to have a user account, and define permissions on a per-worksheet basis. Our security model is based on a standard access control list (ACL) paradigm. A user can only edit a worksheet or cell if included in that CouchDB document's `writers` field (which is a list). The kernels associated with a worksheet should permit the same users to execute code, and ignore requests from everyone else.

CouchDB has a built-in authentication system. It stores accounts and their associated security roles in a special "users" database. This provides a simple way to distinguish between administrators, other existing accounts, and anonymous users.

CouchDB supports several authentication schemes. It implements a form of the OAuth authentication protocol[5]; however, this allows for application-wide access. We want to authorize users on a per-worksheet basis, so this isn't a viable solution. CouchDB also allows users to include their username and password in the URL of each HTTP request. This is convenient for testing purposes, but rather poor security practice.

We decided to use CouchDB's cookie authentication. HTTP cookies are a way for web sites to store information about a user's session [27]. Cookies are stored as text, on the client side (often by a web browser). CouchDB provides a "session" endpoint, from which users can obtain a cookie. Upon receiving an HTTP POST request at this URL,

[5] OAuth is a protocol to allow secure interoperation between web service APIs [7].

with a matching username and password as parameters, CouchDB will supply a session cookie.

This cookie authenticates a user for an amount of time that the CouchDB server defines. It allows users to make authenticated requests to CouchDB without needing to supply their username and password each time. This is important for user experience in Poolside: users must write to the database to perform routine actions like adding, deleting, and evaluating cells.

CouchDB's HTTP authentication mechanism requires that a username and password are sent in plain text. It's rather insecure to transfer this information, or an authentication cookie, over the network. One standard way to prevent eavesdropping on network requests is SSL/TLS (Secure Sockets Layer/Transport Layer Security), a security measure that is the basis of HTTPS. The current release of CouchDB does not support SSL, but it is implemented in the development version [38]. In the meantime, it's possible to direct all traffic to CouchDB through an HTTP proxy that does allow SSL, such as *nginx*.

Web Server

CouchDB stores and serves the entire Poolside notebook: the application itself, and all user data. It acts as both a database and a web server. A current technical limitation of CouchDB makes it a bad choice to host evaluation servers, but it handles most everything else.

CouchApp

A CouchDB database usually contains one or more "design documents". Design documents are special because they contain application logic for a database. CouchDB

looks in design documents for map/reduce views (for querying), validation functions (to prevent unwanted document updates), list functions (for transforming data), and other application code [15]. The code for these functions is stored as strings in JSON.

Like other documents in CouchDB, a design document is stored as a JSON object. It can also store files (HTML, CSS, JavaScript, images, etc.) as attachments. This allows CouchDB to host entire web applications in a single design document. Unfortunately, CouchDB doesn't provide a convenient way to attach many files to a document. Its HTTP API only allows a single attachment per request. The de facto workaround for this problem is a Python package called *CouchApp*.

CouchApp aims to ease the management of CouchDB design documents. It recursively loads a directory from the local filesystem into a design document. This allows developers to split different components of a design document into separate files and folders. CouchApp was very helpful for organizing the various pieces of Poolside's CouchDB code.

HTML Front End

The notebook interface is written using HTML, JavaScript, and CSS. It uses the JQuery JavaScript library at its core, and relies upon a visualization library called *Protovis*.

"URL rewriting" is a CouchDB feature that Poolside uses to render worksheets as HTML. This uses the special `rewrites` field on a design document to simplify an application's URLs. It allows Poolside's users to view a worksheet as an HTML notebook, without needing to know what a list function is.

Interface to Code Evaluation

AJAX Requests

Poolside's web notebook allows users to communicate with an evaluation server via AJAX. Poolside stores the address of an evaluation server as an attachment to its CouchDB design document. The notebook reads this address when the page loads, and directs all code evaluation requests to that URL. The evaluation server reads its own address from the same attachment. New installations only need to change the server's location in one place, which helps with portability.

Poolside's evaluation server accepts HTTP GET requests, so browsers can asynchronously send it requests via AJAX. Each worksheet has its own set of kernels: one for every available language. Individual kernel processes will block while computing, and queue sequential requests. On the other hand, a Ruby request will execute normally during a long-running Python computation. Poolside uses non-blocking AJAX on the client side, so an ongoing computation will not prevent users from modifying other cells. A future implementation might include a fully parallelized version of each kernel, which would allow multiple cells of the same language to execute simultaneously if resources were available.

Cross-Origin Requests

Many web browsers currently restrict cross-domain requests: in most cases, a web page cannot make an HTTP request to a URL on a different server or port. This is a security feature to prevent certain cross-site scripting (XSS) attacks.

Poolside must find a way to deal with this limitation, because it makes requests from CouchDB (the in-browser notebook) to a code evaluation server that runs on its own

port. *JSONP* and *Cross-Origin Resource Sharing* (CORS) are two major ways to make cross-domain requests in a web browser.

JSONP relies on the fact that web browsers allow HTML `<script>` tags to request URLs across domains. An HTML page can load a JavaScript file from a remote server without the browser complaining. This is an effective loophole for making cross-domain requests. A web page can simply insert the result of a cross-origin request into a `<script>` tag, if the server returns content as JavaScript (by setting the `Content-Type` HTTP header to `application/javascript`). The client application will evaluate this data as JavaScript code.

In addition to other request-specific parameters, JSONP queries typically include a callback function. The server will wrap its response data with a call to this function, as demonstrated in the following Python code:

```
>>> print "%s(%s)" % (callback, message)
display({"a": 97, "b": 98})
```

This uses Python's string formatting syntax: substituting in the values of variables `callback` and `message` for each `%s`. These are `"display"` and the string representation of the dictionary `{"a": 97, "b": 98}`, respectively. On the client side, this will call a JavaScript function named "display" with the specified data.

CORS is an alternative to JSONP. It uses a set of HTTP headers to validate cross-origin requests. Each client request includes an `Origin` header, containing the address from which the request originated. Using the `Access-Control-Allow-Origin` header, a server specifies a list of domains from which it will accept requests. This is a whitelist approach: if a client request's `Origin` header is not included in the list of allowed origins, the request will fail. Servers can also specify a wildcard (`*`) to accept requests from all domains.

We initially decided to use CORS in Poolside, but found JSONP simpler to implement.

HTTP Proxying

Nginx is a "reverse proxy", an intermediate layer between a client and one or more servers [23]. It accepts an HTTP request from a client (like an ordinary web server), passes that request to a different server, and then forwards the response back to the client. Nginx solves two problems for Poolside: cross-domain requests, and eavesdropping.

We use Nginx to make CouchDB and the evaluation server accessible from the same domain. CouchDB and the evaluation server still run on different ports, but clients access both of them through nginx. This reverse proxying is enough to satisfy web browsers' same-origin policy for AJAX requests.

Imagine that nginx listens on `localhost:443`, CouchDB on `localhost:5984`, and the evaluation server listens on `localhost:8283`. Clients ordinarily access the notebook CouchApp in CouchDB's domain, port `5984`. Web browsers will consider an HTTP request from CouchDB to the evaluation server (port `8283`) to be cross-origin. This setup requires a workaround like CORS or JSONP.

Nginx provides a simpler solution. It forwards requests from `localhost:443/eval` to the evaluation server (port `8283`), and all other requests from `localhost:443` to CouchDB (port `5984`). The servers to which Nginx is passing HTTP requests will continue to function independently, but they appear to clients to be running on the same port. AJAX requests from one server to the other work properly.

Transport Layer Security

The Transport Layer Security (TLS) protocol is designed to allow secure communication over a network [28]. It is based on the Secure Sockets Layer (SSL) protocol. TLS enables encrypted connections between two agents, e.g., a web server and a client. HTTPS (HTTP Secure) uses TLS as a foundation for secure web browsing. It is a standard way to combat eavesdropping on a network, and forms a critical part of Poolside's security.

A default CouchDB install uses regular HTTP. This is a general privacy issue, because all communication between a user and the CouchDB server passes over the network in plain text. HTTP also presents a special security risk to CouchDB's cookie authentication system, because it sends login information (i.e., username and password) to the server in plain text. This is extremely insecure. Poolside also sends a user's CouchDB authentication cookie in requests to the evaluation server, and plain HTTP leaves users vulnerable to *sidejacking attacks*[6].

Again, nginx is very useful. It can easily be configured to serve data via HTTPS, and then pass regular HTTP requests to CouchDB and the evaluation server. As in the cross-domain request scenario, nginx accepts all incoming client requests and redirects them to the appropriate server. This setup provides a vast improvement in Poolside's security.

In theory, CouchDB should fix our cross-domain request problem with a new "HTTP proxy handler" feature in the upcoming 1.1.0 version [20]. This will allow CouchDB to proxy requests from a URL in its domain to an external HTTP server, effectively fulfilling nginx's current duties. The 1.1.0 CouchDB release will also apparently enable built-in SSL, so it will likely remove Poolside's need for nginx. It's hard to predict when an open source project will make a significant release, so Poolside will be using nginx

[6]Sidejacking occurs when an attacker "sniffs" another user's session cookie over a network, and uses the cookie to impersonate that user.

for the time being.

Code Evaluation Server

Idea

We want Poolside to have computational capabilities. We built an HTTP server to act as a backend computational engine. Clients can make requests via HTTP GET. This evaluation server currently provides "kernels" in two dynamic, high-level programming languages: Python and Ruby. Both are free and open source. Poolside is designed, with a few limitations, to allow remote execution of arbitrary code. This feature has many associated security issues.

Dynamic High-Level Languages

Programming languages are sometimes split into a pair of categories: high-level vs. low-level, dynamic vs. static, strongly-typed vs. weakly-typed, compiled vs. interpreted, and so forth. An additional distinction is "developer time" vs. "computer time". This compares the number of hours that it takes to write a program, and how long it will ultimately take to execute, across languages.

For Poolside, we initially chose to use dynamic languages; the codepad web application compiles and runs programs in C and C++. We want users to be able to run an interpretive session, to update variables over the course of multiple commands.

We chose languages that emphasize developer time over execution speed. Our current architecture is designed for quickly trying out code, which is why the evaluation server imposes a limit on CPU time for each kernel. Future work on Poolside might enable

resource-intensive scientific computing, by tapping into cloud-based hardware from a vendor such as Amazon [14].

Server Details

Poolside's evaluation server is written in the Python programming language, version 2.7. Python provides a "BaseHTTPServer" module in its standard library, and we used this as the foundation for Poolside's evaluation server. There are Python packages for building scalable web servers, such as Twisted and Tornado; however, we wanted to minimize external dependencies in our initial implementation.

Dynamic Language "Kernels"

Each kernel is a standalone script that runs as an infinite loop. It waits for input, evaluates the input, and then prints the result of its computation. The server creates one kernel per available language, for each worksheet. Python and Ruby are currently enabled, so there are two kernels for every worksheet.

Each kernel runs in its own process. The evaluation server creates these subprocesses, and communicates with them via *anonymous pipes*. The *standard input* and *standard output* streams of each kernel process are redirected back into the parent (server) process[7]. Standard input is the primary source for programmatic keyboard input; standard output is the location to which a program's main output gets printed. Every kernel is a read-eval-print loop (REPL): a `while` loop that reads from standard input, evaluates that text as code, and prints the result to standard output.

This form of inter-process communication is general to a variety of programming lan-

[7]The ANSI (American National Standards Institute) standard for the C programming language defines "standard input", "standard output", and "standard error" streams [18]; Python and Ruby follow this convention.

guages. One advantage of our approach is that it is extensible: Poolside's evaluation server communicates with kernels in a language-neutral way. This will make it fairly simple to add new languages in the future.

The evaluation server communicates with the kernels not in purely asynchronous fashion, but rather by alternating input and output messages. Poolside uses *line buffering* to separate messages sent over standard input and standard output. Each communication between the evaluation server and a kernel is delimited by a newline character. This is a simple but potentially fragile approach, because incoming Python code such as

```
for i in range(3):
    print i
```

will get broken into two messages: `for i in range(3):` and `print i`. The string representation might be `"for i in range(3):\n    print i"`[8], and the kernel process will interpret the part after the first newline as a separate command.

Poolside's current solution to this problem is to temporarily replace each newline character with a character that users are exceedingly unlikely to input. This contortion allows Poolside to send data over line-buffered standard input and standard output, while preserving all original newlines. We've chosen the Unicode character represented by the UTF-8 value `FFFF`[9]. This *code point* is guaranteed by the Unicode standard to be a "noncharacter", reserved for use within applications.

Adding New Languages

There are a variety of open source dynamic, high-level programming languages. Our initial implementation is limited to Python and Ruby; however, other languages, like

[8] Newline character sequences vary across platforms. The conventions are `\n` on Unix, `\r\n` on Windows, and `\r` on Macintosh [1].

[9] Unicode is a standardized text encoding.

Perl and R, would be easy to add. For the sake of simplicity, we start with only two.

In order to work as a kernel, a language must be able to evaluate strings as code *at run time*. This feature is common to most languages that we call “dynamic”. Compiled languages, such as C and Java, generally cannot do this. The current implementation also requires a string substitution function, as explained above. Replacing instances of a particular character in a string is our method of reconciling line-buffered input/output streams with the fact that newline characters are significant in Python.

A kernel process reads from standard input, and will wait (block) until a newline-terminated string appears. This string, which has a Unicode noncharacter in place of all its original newlines, must next be restored to its initial state. The kernel then evaluates this sanitized input, and finally prints the result to standard output.

Each kernel maintains a namespace, saving variables from one computation to the next. This is an important feature, because it’s difficult to program without variables.

Security

It seems like a bad idea to allow anyone on the internet to execute any command on one’s computer. Both Python and Ruby provide access to system commands. This allows users to programmatically create and delete files, consume system resources such as memory and network bandwidth, and do other potentially nefarious things. While system administrators can hope that users will be responsible, history suggests that this is a somewhat naive security policy.

Fortunately, there are tools for constraining users and processes. We will focus on features available in Linux and other Unix operating systems. We use two such security mechanisms, *chroot jails* and *resource limits*, in Poolside. Other platforms may have different approaches, but we will not discuss them here.

Chroot jails

The Unix directory structure is a tree. Each directory has a parent and zero or more children. The child nodes need not be directories: they can be text files, for example. A special case in the directory hierarchy is the *root* node, the base (top) of the tree. There is nothing above the root directory.

Unix provides a `chroot` command to change the apparent root directory of a process. This effectively creates a "ceiling" for the process, preventing it from accessing directories above the current one. The `chroot` command can be used as a security feature, to limit a program's view of the filesystem. This is called a "chroot jail".

A properly-implemented chroot jail is a secure way to limit filesystem access. Breaking out of a chroot jail generally requires root (superuser, administrator)[10] privileges. Successfully calling `chroot` requires root privileges, so a common pattern for creating chroot jails is to run the chroot command as the root user and then drop privileges. Another Unix command, *setuid*, can be used to become an unprivileged (non-root) user.

In order to function properly, a chroot jail typically must contain several files. A program such as Python does not run in a vacuum: it makes use of existing libraries. The Unix command `ldd` prints the libraries on which a program depends:

```
[david@myhost ~]$ ldd /usr/bin/python2.7
    linux-gate.so.1 =>  (0xb7765000)
    libpthread.so.0 => /lib/libpthread.so.0 (0xb77c1000)
    ...
    libm.so.6 => /lib/libm.so.6 (0xb7562000)
    libc.so.6 => /lib/libc.so.6 (0xb7400000)
```

[10]The *root directory* and *root user* are independent concepts with the same name.

A Python process running inside of a chroot jail will only function properly if the libraries mentioned by `ldd` are present. Following the above example, a chroot jail with `/tmp/jail` as its root should have a copy of the *libpthread.so.0* library file at `/tmp/jail/lib/libpthread.so.0`. Python also depends on numerous modules from its standard library, such as *random.py*, for a wide array of functionality. Python must be able to find these files inside of the chroot jail.

The last step in creating a chroot jail is setting file permissions. Processes running inside the jail should generally not be able to modify important files. The root user and group can own all of the files in a chroot jail, thereby preventing unprivileged users from modifying the filesystem. As long as the initial process drops privileges properly, the files in a chroot jail should be isolated and safe from user interference.

Per-Process Resource Limits

Operating systems have access to finite amounts of resources such as memory and processing power. When these resources are unavailable, programs might fail to function properly. In order to mitigate this risk of a system becoming unresponsive, an administrator can impose "resource limits". There are a few tools for doing this on Linux, including the *PAM* (Pluggable Authentication Modules) project, the *ulimit* utility, and the POSIX *setrlimit* function. *PAM* is more concerned with authenticating users at the operating system level than enforcing resource limits. We don't want to include a shell inside the chroot jail, and *ulimit* is part of the *bash* shell. We therefore chose to use *setrlimit* in Poolside.

The Unix *setrlimit* command sets per-process limits on system resources such as bytes of memory and seconds of total CPU time. There are two constraints for each resource: a hard limit and a soft limit. A hard limit is the maximum amount of a specified resource that can become available to a particular process; it is a ceiling for the soft limit, and one

that only a user with administrative privileges can increase. The soft limit represents a working restriction. On user attempts to surpass this rule, the operating system will cause the associated low-level operations (e.g., system calls to allocate memory and open files) to fail. If a process exceeds its cap on CPU time, the operating system will kill it. [12].

The Python 2 standard library contains a module called "resource", a thin wrapper around *setrlimit* and the associated *getrlimit* function. Using this module, Poolside currently limits each kernel process's total CPU time, memory, number of child processes, and number of open file descriptors. The limits are somewhat arbitrarily chosen; resource limits should generally depend on a particular system's available resources. The evaluation server sets resource limits for each kernel when creating its subprocess, so the *setrlimit* logic is contained in one place.

Interface to CouchDB

The evaluation server relies on CouchDB for two things: to discover the port on which it should listen, and to authenticate requests.

When starting the evaluation server, an administrator must specify the URL of an attachment on a Poolside instance's design document. This JSON attachment contains the address (host and port) at which the server will run. The Python program that runs the server requires this as a command line argument. The server information, used by both the evaluation server and the notebook frontend (to make code evaluation requests), is stored in only one place. Alternative solutions, such as requiring the evaluation server to run on the same machine as CouchDB, or "hard coding" the information in multiple files, are less appealing.

The evaluation server utilizes CouchDB's authentication features as a security measure. Client HTTP requests must include a valid CouchDB cookie, or they will be rejected.

The server can also check that a user is authorized to write to the worksheet with which a requested kernel is associated. Poolside currently stores a “writers” field on each cell and worksheet document in CouchDB. The evaluation server can use this information to prevent unauthorized users from modifying kernels.

CouchDB External Processes

CouchDB has a feature called “external processes” that allows administrators to write extension programs to CouchDB, in their language of choice. CouchDB provides a URL for every service, within each database. It forwards all HTTP requests of those URLs to the external program. CouchDB passes in a JSON object that includes the user’s request itself, information about the database, and an authentication object. The last feature is especially useful, because it simplifies the authentication of CouchDB users in a separate process.

Unfortunately, there is a big problem with regard to scalability. CouchDB communicates with external processes via standard input and standard output. These streams cannot be read concurrently, so CouchDB must handle each request in a blocking fashion. This means that the external process can’t accept a new request until the previous one completes. For a multi-user server with potentially long-running computations, this will not work. We therefore run Poolside’s evaluation server as a standalone process.

Results

Poolside is currently an incomplete implementation, and I'm not sure that it's ready for production. The notebook interface can use quite a bit of polishing, and the code evaluation server could have better security. A visualization framework is in place, but the available tools are extremely limited. Despite the obvious room for improvement, I believe that Poolside might already be able to help users share analysis and data online. The note-taking and computational aspects seem to work properly, and it's a functional web application.

In its current state, I suspect that Poolside might be most useful in an educational computer lab setting. User accounts on lab computers often either lack the privileges to install software, or they are unable to save changes between sessions (i.e., data is lost when the system shuts down). Poolside allows users to immediately begin writing code, rather than requiring an administrator to intervene. It saves data across sessions, and is accessible from a web browser. Poolside could be a tool for helping students learn to program.

Future Work

Poolside is an open source project, currently hosted online by *GitHub* [2]. All of Poolside's source code is available there; it's still a bit rough around the edges.

This thesis concludes by listing a few ways in which the software could be improved. These tasks are not too difficult technically; however, they remain unfinished due to time constraints.

Poolside should be able to make better use of its cell data. Each cell is stored as a

CouchDB document, and thus as JSON. There should be a way to create data pipelines, as in Galaxy or the Python application *Pypes* [9]. Users could use the output of one cell as the input of another, chaining results together. It might also be useful to directly load a cell's JSON output into a kernel, without needing to recompute the result each time.

Data visualization is another realm in which Poolside can improve. It shouldn't be too hard to allow users a choice of visualizations on the client side (with the Protovis JavaScript library). At time of writing, there is only one visualization available, more of a "proof of concept" than a feature.

Earlier versions of Poolside allowed server-side graph generation, using the Python *matplotlib* plotting library. Images could be saved to CouchDB as attachments to cell documents. For large output data sets, a matplotlib-enabled evaluation server could send back a graphical representation of data, instead of an extremely long string.

Python's BaseHTTPServer doesn't appear to be widely used in production environments. Poolside's evaluation server might be more scalable if we used more specialized Python server software such as *Tornado* or *Twisted*.

Poolside would do well to have more language kernels, e.g., the R statistical programming language. This might draw more users, and it generally makes the software more flexible.

Finally, there are many small improvements that could be made to Poolside. These include the enforcement of CouchDB user permissions on kernels, and better cleanup of "stale" kernels.

Software is rarely, if ever, completely done. Poolside is certainly a work in progress, and hopefully it will continue to be developed. Regardless of its current usefulness, Poolside still has plenty of room to grow.

Appendix

Past Architectures

Over the course of the last year, this project has gone through a number of different architectures. All of these prior designs have influenced Poolside's current state. Python and CouchDB have consistently been part of our various achitectures, but many other components have come and gone.

Initial Architecture

Our preliminary architecture was based on Galaxy, and Python bioinformatics software called *pygr* [8]. Pygr supplies several convenient data structures for working with genomes. It also provides a global namespace for data, called "worldbase": a unified interface to files that might be on different servers. We intended to incorporate an HTML notebook into Galaxy as a "tool", with pygr's worldbase as the data storage backend.

Our first actual implementation was based on worldbase and SageNB, the web frontend to Sage. We planned to plug this into Galaxy, allowing users to track their analysis pipelines via Galaxy's "workflows". We had wanted to use CouchDB for the notebook, but we didn't know about CouchApp yet.

We found a few issues with this approach. We had problems passing data between Galaxy and SageNB, with or without worldbase. Sage is also a fairly large software dependency. So, we decided to build our own notebook.

Building a New Notebook

We decided to use Django, an open source Python web framework, as the base of our notebook. We built the frontend JavaScript with *Pyjamas*, a Python-to-JavaScript compiler. I was not particularly familiar with JavaScript, and this allowed me to write HTML/JavaScript "widgets" in Python. The frontend made requests to a Python server via the JSON-RPC (remote procedure call) protocol.

This Django-based architecture had a lot of different components. Eventually, we came across CouchApp and decided that hosting the entire application in CouchDB would simplify our task.

Yet Another Notebook

Storing the HTML, CSS, and JavaScript in CouchDB worked well. We finally found a good way of structuring the notebook, but the computational server continued to be a problem. We decided to base our architecture on IPython's development version. Its ZeroMQ message passing system, with a JSON API, seemed to fit in well with CouchDB. IPython has a lot of features, but we weren't using most of them. The development branch was in a state of flux at the time, and changes to IPython's core occasionally broke our code.

To connect Poolside's notebook to the evaluation server, we tried using CouchDB's external process mechanism. As described earlier, this approach is unusable for us: the server handles requests in a blocking fashion. In what now seems like a bit of an overreaction, we decided to try a radically different approach.

Standalone Server and an HTML5 Frontend

Lacking CouchDB's external process feature, we chose to use an open source Python web server called *Tornado*. I also decided to utilize an HTML5 protocol called *WebSockets*. This allows a browser and a server to open a persistent connection, and send data bidirectionally. One nice side effect of a WebSocket-based approach is that the server can react to client disconnection events. This is harder to do with traditional AJAX or long-polling setups.

Unfortunately, WebSockets are not well-supported by most web browsers. The current generation of HTML-enabled browsers does a reasonably good job, but WebSockets simply don't work in older versions. There is a technique that uses *Flash* as a fallback, but there are also security issues associated with the current draft of the WebSocket protocol [30].

We decided to make one final major architectural shift, with an emphasis on simplicity. We switched from Tornado to a server derived from Python's standard library, and from WebSockets back to a traditional AJAX approach.

Current Architecture

At the core of Poolside's current architecture is a CouchApp. All of the notebook's HTML, CSS, JavaScript, and image files are stored in CouchDB. When a user evaluates a cell as code, the notebook makes an AJAX request to the evaluation server. This server is based on the BaseHTTPServer module in Python's standard library. It can handle multiple requests in an asynchronous fashion, and generally seems to work as intended.

Bibliography

[1] 2. Built-in Functions — Python v2.7.1 documentation. http://docs.python.org/library/functions.html.

[2] dwyde/poolside. https://github.com/dwyde/poolside.

[3] Frontpage - IPython. http://ipython.scipy.org/moin/.

[4] Google Wave Federation Protocol. http://www.waveprotocol.org/.

[5] joewalker/cockpit. https://github.com/joewalker/cockpit.

[6] JSON. http://www.json.org/.

[7] OAuth Community Site. http://oauth.net/.

[8] pygr: Scalable bioinformatics interfaces in Python. http://code.google.com/p/pygr/.

[9] Python Package Index : pypes 1.1.0. http://pypi.python.org/pypi/pypes/.

[10] Sage Notebook. http://nb.sagemath.org/.

[11] Sage: Open Source Mathematics Software. http://sagemath.org/.

[12] *setrlimit(2) - Linux man page*.

[13] Tomboy Online (Development). https://edge.tomboy-online.org/.

[14] Amazon.com. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[15] J. Chris Anderson, Jan Lenhardt, and Noah Slater. Design Documents. http://guide.couchdb.org/draft/design.html.

[16] J. Chris Anderson, Jan Lenhardt, and Noah Slater. The Power of B-trees. http://guide.couchdb.org/draft/btree.html.

[17] Renzo Angles and Claudio Gutierrez. Survey of Graph Database Models. *ACM Computing Surveys*, February 2008.

[18] ANSI. The C Standard Library. http://www.utas.edu.au/infosys/info/documentation/C/CStdLib.html#stdio.h.

[19] ZOHO Corp. Zoho API Developers Guide. http://apihelp.wiki.zoho.com/Zoho-API-Developers-Guide.html.

[20] Paul Davis. Re: Unable to use the HTTP proxy handler in the new externals API. http://mail-archives.apache.org/mod_mbox/couchdb-user/201103.mbox/browser.

[21] The Economist. Data, data everywhere. *The Economist*, February 2010.

[22] International Organization for Standardization. Database Language SQL. http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

[23] The Apache Software Foundation. mod_proxy - Apache HTTP Server. http://httpd.apache.org/docs/2.0/mod/mod_proxy.html#forwardreverse.

[24] The Eclipse Foundation. Orion - Eclipsepedia. http://wiki.eclipse.org/Orion.

[25] James Gao. IPython HTTP frontend. http://mail.scipy.org/pipermail/ipython-dev/2010-October/006818.html.

[26] Google. Status of Google Wave - Google Wave Help. http://www.google.com/support/wave/bin/answer.py?answer=1083134.

[27] IETF Network Working Group. RFC 2109: HTTP State Management Mechanism. http://www.ietf.org/rfc/rfc2109.txt, February 1997.

[28] Internet Engineering Task Force Network Working Group. RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2. http://tools.ietf.org/html/rfc5246.

[29] Steven Hazel. about - codepad. http://codepad.org/about.

[30] Chris Heilmann. Websocket disabled in Firefox 4. http://hacks.mozilla.org/2010/12/websockets-disabled-in-firefox-4/.

[31] Infinote. Infinote. http://infinote.org/.

[32] Martin Manns. Pyspread. http://pyspread.sourceforge.net/.

[33] MediaWiki. Extension:GraphViz. http://www.mediawiki.org/wiki/Extension:GraphViz.

[34] MediaWiki. Extension:sparqlextension. http://www.mediawiki.org/wiki/Extension:SparqlExtension.

[35] Microsoft. Office Web Apps. http://office.microsoft.com/en-us/web-apps/.

[36] American Physical Society. “What is Science?”. http://aps.org/policy/statements/99_6.cfm.

[37] UnaMesa. Tiddlywiki - a reusable non-linear personal web notebook. http://www.tiddlywiki.com/. HostedOptions section.

[38] CouchDB Wiki. How to enable SSL - Couchdb Wiki. http://wiki.apache.org/couchdb/How_to_enable_SSL, July 2010.

[39] CouchDB Wiki. Introduction to CouchDB Views. http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views, January 2011.