

Saint Martin's University: CSC 510

Deployment of a Jupyterhub Server

By Alexander Benson, Matt Dunaway, and Taylor Johnson

Table of Contents

Introduction.....	2
Motivation and Original Contribution	3
Background and Literature Review	3
A Hybrid Approach of Compiler and Interpreter.....	3
A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering (Fangohr, 2004) ..	4
Language Considerations in the First Year CS Curriculum (Blake, 2011).....	4
Interactive Workflows for C++ with Jupyter (QuantStack, 2019).....	5
Integration of Dotty in Jupyter notebooks (Gehrig & Martres, n.d.)	6
Proposal Design and Implementation	7
Proposal	7
Implementation	7
Amazon Web Services	7
Home Server	9
Testing and Evaluation	10
Summary	11
References.....	12

Introduction

When our team began on the midterm report, we researched a variety of programming topics in recent and current research. We talked through many interesting topics of research dealing with topics such as advances in SCADA technology, centralized industrial IT infrastructure, and hybrid approaches to compiling and interpreting programming languages. We also explored research covering features and programming techniques of programming languages such as C, C++, PERL, JavaScript, and Python. After performing our literature review for the midterm report and extensive discussion within our team, we decided to pursue a project that would advance our own understanding of programming languages and could also be used as a tool in computer science (CS) education for beginner CS students and teachers. This project's goal was to successfully build a JupyterHub server that could interpret multiple programming languages on one platform. This would provide a learning tool so users would understand the differences between languages that are traditionally compiled versus interpreted. As we will explain in greater detail in this final report, the JupyterHub build was a success. The virtual server platform and the programming language kernels were integrated into one hub that our team was able to run multiple languages on. These languages were taken directly from code used in CSC 510 lecture and homework to aid our understanding of current programming topics simultaneously, allowing us to view the topic in other languages in one notebook.

Motivation and Original Contribution

The parameters of our assignment asked for a review of at least ten conference or journal papers related to programming languages and write a summary describing the research in five of them. The team then needed to produce either a proposal for a project or a plan to review relevant research on the topic chosen as the subject of the report. After reviewing over a dozen journal papers related to current research in programming languages, we chose five that we thought were the most interesting and relevant for the curriculum of CSC 510. This research revolved around the education and understanding of taught programming languages in a beginner curriculum. More specifically, each of us had some portion of our research dealing with the compilation and interpretation of programming languages. After much discussion on the complexities of the subject, we decided on a project that could assist students in their pursuit of further understanding the differences between compiled and interpretive languages. The motivation of our final project came from the research of our team member, Alexander Benson. He performed research on programming language education in a collegiate environment as well as the implementation of the JupyterHub server as an education tool. His documentation came from the Jupyter website, <https://tljh.jupyter.org>.

Background and Literature Review

A Hybrid Approach of Compiler and Interpreter

Today, most programs are written in a high-level language and then translated to machine code by a compiler or an interpreter. The push for interpreted languages is in demand due to their simplicity but has increased CPU requirements for execution (Singh, 2012). The interpreted code is also larger than needed for execution, and there is no way to minimize code. To explain further concepts, a description of what an interpreter and compiler do is helpful. Compilers translate high-level language in phases. Lexical analysis forms tokens from the program text. Syntax analysis takes the tokens and organizes them into syntax trees. Type checking determines if the tokens of the syntax tree are consistent and if errors are present. Intermediate code generation translates the checked code into a symbolic machine code. Machine code generation turns the symbolic code into a machine code assembly language. Assembly code is finally turned into binary code. The interpreter performs some of the same phases, including lexical analysis, parsing, and type checking, but executions and expressions are performed directly on the syntax tree. The interpreter must return to the syntax tree multiple times for specific data and, thus, is executed slower than a compiler. Conversely, the interpreter is easier to write and has greater compatibility with different machines.

The proposed hybrid approach starts with the compilation process to produce the intermediate code. The intermediate code is then interpreted rather than compiled. One of the most compelling examples of this was using Google Closure Compiler, which compiles from JavaScript to parse, type check, and edit code back to a minimized Java format (Singh, 2012). After this ran, an interpreter performed at a similar speed to a compiler. A proposal was then put forth to package the closure compiler method and a simplified interpreter to make a *compreter*. This would share

the benefits of both translation methods of speed simplicity and portability into one translation apparatus.

A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering (Fangohr, 2004)

The use of computing languages in engineering is typically for solving a problem. In this context, algorithms and pseudocode implementations are common. The paper explores an implementation of the composite trapezoidal rule (a numerical integration algorithm). The article discusses the languages used in an anecdotal tone. Experimentation was performed with different classes; however, the data collected was qualitative, and no quantitative information was collected. The writers relate their experience teaching the languages, common frustrations experienced, and syntactic structures in the different languages that enable learning or frustrate beginners.

Some of the common problems with C cited make for plausible pitfalls while others sound preventable and unrelated to C in particular: Indentations and scopes do not agree; missing semicolons and curly braces; and the passing of the wrong type to function calls/printing with the incorrect format identifier. This last point makes more sense as a struggling point when the authors highlight that passing the wrong type in C will not necessarily result in an error being thrown. However, it can affect the numerical results. Given the complicated nature of some algorithmic implementations in numerical analysis (such as Runge-Kutta methods), it is reasonable that students might struggle with the permissive nature of C and the lack of visibility into variable values.

The authors state that MATLAB typically presents less of a challenge for their students given that its error detection provides high-quality feedback, and its typing can be more intuitive. The authors do highlight that MATLAB is best managed using smaller global functions. However, these functions must be implemented in individual files (one per function) and stored in the same directory as the main program. For this reason, function management can be difficult for many new students. Related to this is that the function name, when called, is determined by the file name. Students can name the function something other than the file name and struggle to find the source of their error when running the main function.

The authors note programming similarities in Python to MATLAB that lend it some of the advantages of the latter. Although there is some significance placed on the lack of curly braces (this appears to have been a central sticking point for some), the authors highlight that the visual clarity of controlling the function strictly through indentation is preferable to their students. Function definitions in Python are intuitively implemented, and typing has been found to be more intuitive.

The authors conclude that Python is the preferred language for training and teaching engineers for reasons related to the intuitive nature of programming, the economics of a free language vs. the paid MATLAB license, and Python's OS-agnostic implementation.

Language Considerations in the First Year CS Curriculum (Blake, 2011)

The author outlines the history of programming language selection in the field of computer science. There are many familiar points made by the author, reminiscent of similar debates in most forms of education. At the crux of the matter is the degree to which computer science

should teach students to program without teaching students to become programmers, a job function that the authors consider philosophically separate from computer science. In outlining the history of computer science education, the author hints that their colleagues may have unduly caved to industry pressure to teach the latest language of the day. In teaching a particular language, students potentially become wedded to a strict syntax or programming paradigm, failing to understand or appreciate the broader purpose behind a language or paradigm. The author states that they began with Ruby and appreciated that scripting languages demand fewer complicated topics before writing code. However, the author criticizes the body of work outlining the use of Ruby, stating that there is an extensive body of literature related to programming in Ruby, but they focus on the writing process and do not dwell on the programming concepts needed for an introductory computer science course. Java was attempted as an alternative. However, students found Java to be unintuitive and difficult to transition to; Ruby was easier to learn and retain.

Groovy was attempted as a bridge to Java, given its ability to interact with Java while providing a more intuitive syntax and structure. There were some problems with Groovy identified: Java and Groovy have different typing methods, different syntax, the number of control structures in Groovy is so large as to be detrimental, and Groovy does not respect private access modifiers. For the reasons outlined above, Scala is the preferred language for educating first-year computer science. The language is considered sufficiently like Java to provide familiarity with the language, and Scala supports interoperability with Java. Scala's interactive shell and scripting capabilities were found to be helpful educational tools, and at the end of the semester, students experienced a reduction in the difficulty of transitioning to Java.

Interactive Workflows for C++ with Jupyter (QuantStack, 2019)

In the view of the authors, interactive computing/interactive programming is essential in productive programming. C++ lacks a strong history of interactive computing, making C++ challenging to teach. "The goal of Project Jupyter is designed to provide a consistent set of tools for scientific computing and data science workflows" (QuantStack, 2019). As such, Jupyter Project derived tools emphasize interactivity and data visualization. The Jupyter stack is built upon the kernel. Given the range of kernels and the breadth of available languages, the Jupyter Stack could be cautiously referred to as language-agnostic. Most of the languages rely on a C API embedding into an application. To simplify the kernel building process, the authors wrote the Xeus implementation.

Xeus is an implementation of the Jupyter kernel built using C++. It is a library simplifying the creation of kernels, not a kernel. Xeus-Cling is a C++ kernel already in use at CERN and, through JupyterHub implementations, used to instruct students at Paris-Sud University. Since this is a Jupyter product, several tools, such as dynamically edited plots, tensors, and widgets, exceed the scope of this paper. All the same, specific libraries the authors point to libraries in the xtensor ecosystem that could be of use to C++ programmers. Of particular interest is the [xtensor-blas](#) library, providing linear algebra functions to C++. The inclusion of linear algebra libraries within C++ nicely unites the speed of C++ with the increasing significance of matrix algebra within computing.

Integration of Dotty in Jupyter notebooks (Gehrig & Martres, n.d.)

According to the author, Dotty is the experimental form of Scala 3.0. Given work by other authors (Blake, 2011), Scala might be an ideal language for first-year computer science students. Integration with the Jupyter Notebook environment represents an opportunity to unify an educational programming language with an interactive workspace. The author states that it is now possible to make Jupyter kernels capable of supporting notebooks in any language. This paper outlines the work performed to create a Jupyter kernel of Dotty.



Figure 1: Some of the libraries utilized and their arrangement (Gehrig & Martres, n.d.)

Early complications the author encountered were a lack of clear protocols for multithreading/parallelism in Jupyter. While users are not permitted to run cells concurrently, this limitation extends to the IPython kernel, which is the foundation of the Jupyter environment. While distinct, this is related to the difficulties of remotely interpreting code. Initially implemented, the author found that the remote call to the interpreter was sending code then waiting for a returned value and printing it. This solution was deemed sub-optimal given that it limited any debugging functionality. The answer was to increase communication between the server and the client. The server immediately reports if it has initialized the thread and outputs a boolean value stating whether the thread was still executing code. The appeal of this solution lies in the increased integration of server-client performance, enabling improvements such as interrupting and a user interface that tracks the program status more reliably.

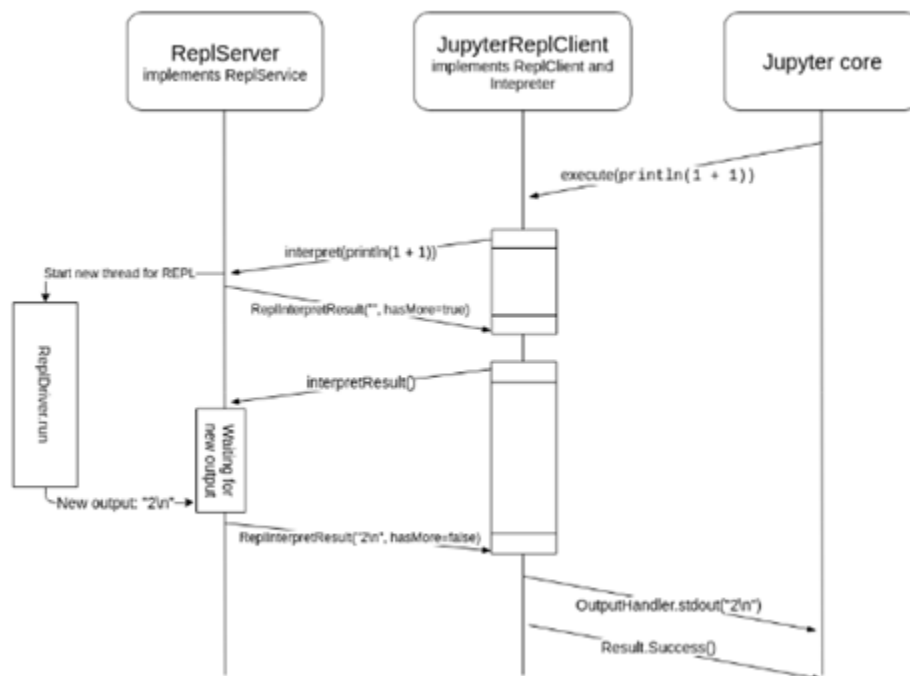


Figure 2: Communication Protocols within the Dotty/Jupyter Implementation

Proposal Design and Implementation

Proposal

The team proposed establishing a JupyterHub Server (*The Littlest JupyterHub — The Littlest JupyterHub v0.1 Documentation*, n.d.) capable of providing interpreted implementations of many programming languages. The purpose was to highlight the difference in interpreted vs. compiled languages. There were several anticipated outcomes: students interacting with the notebook will have a firmer grasp of the distinctions implied by the terms “compiled” vs. “interpreted,” students will have an opportunity to interact with traditionally compiled codes using an interpreted environment, the environment will allow students to switch between languages with minimal friction. There were known limitations to the project as planned: the Jupyter environment would not provide an opportunity to compile code in any convenient or straightforward way; cross-language communication would be limited; the number of JupyterHub users would be small; the languages supported would primarily be C++, Scala or Dotty, Java, potentially SQL, and C# if kernel implementations could be found. Jupyter can support many more languages than those outlined here, however many of these languages are supported by default, such as Python, or R. Python and R are also implemented as interpreted languages exclusively, meaning that the educational barrier for either may not be as high as C++.

Implementation

Amazon Web Services

A prototype server was built on an Amazon Web Services (AWS) free-tier instance. The steps taken were to launch an EC2 instance using an Ubuntu 18.04 image, selecting minimal performance specifications (single-core CPU, 1GB ram, 30GB storage). The instance is classified as a t2.micro by AWS. The user data of the server was modified to initialize the server with a JupyterHub install. The code below is the user data used:

```
#!/bin/bash
curl -L https://tljh.jupyter.org/bootstrap.py \
  | sudo python3 - \
  --admin abenson mdunaway --plugin git+https://github.com/kafonek/tljh-shared-directory --
showprogress-page
```

Unfortunately, the “--showprogress-page” keyword argument failed to enable the desired progress page. It is suspected that this feature is no longer included in JupyterHub or may be enabled without accepting a keyword. The shared directory, “~/scratch”, has been implemented. Future implementations will try to use a more indicative name for the shared folder.

Rules were set to enable the use of SSH, HTTP, and HTTPS protocols on ports 22, 80, and 443, respectively. This was achieved by creating a security group with the necessary settings. After this, the instance was launched. The team had to wait while JupyterHub was installed on the server. This took several minutes.

Once installed, passwords were selected, and the team logged in. At its base state, the JupyterHub instance only had a Python kernel installed. Kernels were added from this point. However, the process highlights a gap in the Jupyter Notebook documentation from an administrative perspective.

Ideally, the process for adding a kernel to JupyterHub should be:

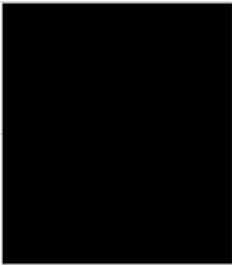
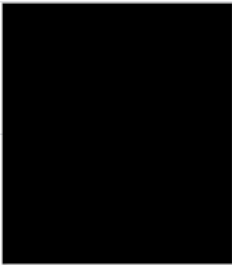
1. Create a virtual environment for the kernel.
2. Activate the virtual kernel.
3. Install the kernel as specified, typically using the command: “sudo -E conda install <kernel>”. “sudo -E” commands JupyterHub to install the kernel globally, aka for all users.
4. Deactivate the environment.
5. Create a reference to the kernel within kernelspec, found using the command “jupyter kernelspec list”.

It is unreasonable to expect users to navigate virtual environments and activate them on their own. Even if users are trained to search for available virtual environments, activating them represents an unnecessary burden on the user. Unfortunately, the JupyterHub documentation does not clearly outline a workaround enabling the critical step 5 shown above. As such, all kernels have been installed in the base environment, which is probably responsible for some of the unstable behavior which will be mentioned later.

C++ is the most popular language covered by CSC 510. As such, adding a C++ kernel took priority. Xeus-Cling is a C++ kernel implemented for the Jupyter Notebook environment. As such, it behaves akin to an interpreted C++ environment. Xeus-Cling is installed via the command “sudo -E mamba install xeus-cling -c conda-forge”. The use of the “mamba” command is essential; for reasons that are not obvious, the conda-forge library does not install Xeus-Cling reliably. Xeus-Cling is easily installed if the correct command is used and performs well once the eccentricities of an interpreted C++ language are understood.

From this point, other kernels were installed, such as Java and JavaScript, however only the Java kernel has been tested on Java code to date. Similar to C++, Java is treated as a strictly interpreted language by Jupyter, and accommodations must be made.

The AWS server’s IP address and HTTPS certificates were directed to this website managed by a team member: hub.pluralbenson.me. As discussed below, an old workstation has been used to replace the AWS server. The URL now points to that server and, after a few hours of downtime, HTTPS has been enabled, see Figure 3.

Host name [?]	Type [?]	TTL [?]	Data [?]
pluralbenson.me	A	1 hour	
pluralbenson.me	AAAA	1 hour	
hub.pluralbenson.me	CNAME	1 hour	calc.pluralbenson.me.
www.pluralbenson.me	CNAME	1 hour	pluralize-abenson.github.io.

Google Workspace

pluralbenson.me/MX, pluralbenson.me/SPF, and 2 more

Dynamic DNS

calc.pluralbenson.me/A

Figure 3: The Google Domains Configuration of the Server and Domains

Home Server

The team could not count on a static IP for the purpose of hosting a web server. For this reason, dynamic DNS and Certbot were required. Dynamic DNS is a service enabling a server on a non-static IP to periodically update its IP with the Domain Name Server. Certbot periodically updates the certificates enabling HTTPS. The local SSH, HTTP, and HTTPS ports had to be forwarded through the router. Once this was done, JupyterHub migrated from a 1GB RAM, 30GB Storage AWS server with a single core CPU server to an 8GB RAM, 200GB Storage, quad-core server.

Testing and Evaluation

Testing took two forms: testing user accounts and testing the kernel functionality. User account testing was relatively straightforward; an account with the username “test” was created, and any changes to the server were checked against the test user to verify that the desired effect had taken place. Kernel functionality was more difficult to test.

For a kernel to be deemed successful, it needed to meet the necessary syntax and grammatical behaviors of the language with a minimal number of adjustments. As mentioned above, interpreted C++ is not a perfect replica of its compiled equivalent. For example, code snippets:

```
#include <iostream>
using namespace std;
```

and

```
int main()
{
}
```

must be stated in separate cells. The main function also does not run by default because interpreted languages allow for single-line commands and C++ respects this convention. As such, defining the function “main()” is identical to defining any other function. If the user wishes to run main, they must call it below the function like so:

```
int main()
{
}

main()
```

The C++ kernel must be the C++ 14 kernel. There is no obvious reason why this is so or why the Xeus-Cling designers would ship C++ 11 and C++ 17 kernels that are strictly for the Jupyter backend. It was originally thought that the kernel required regular reinstallation; “cout” commands have thrown errors that are not seen when the same code is run on a traditional compiler. These errors were once fixed by reinstalling Xeus-Cling. The issue was due to compiling errors: if, for any reason, the interpreter encountered an error at execution, the kernel required a restart. Failing to restart the kernel would result in the “cout” behaviors originally thought to be a complication with the installed kernel. Restarting the kernel clears all environment variables from the notebook, so the user must remember to run any cells that import important libraries and namespaces.

Java's behavior deviates in a very similar manner. Defining a class is not the primary means of controlling flow. Instead, the interpreter favors an approach that could be described as "Python-flavored Java." The Java kernel appears to be more stable than the C++ kernel. No restarts have been necessary to fix unexpected behavior.

Summary

In conclusion, our final project goal of setting up a JupyterHub Server in the AWS virtual server and setting up an interpreted implementation of multiple languages was successful. There were a few complications in the installation of language kernels and a learning curve for the interpreted version of each language. Languages with histories as strictly compiled, such as C++, deviated more from their compiled formats and involved more adjustment than hybrid languages. Overall, JupyterHub was stable enough that multiple languages could be used in the base environment. Improvements to JupyterHub in the future should include clearer support for multiple virtual environments. As such, it could evolve into a learning tool allowing users to understand the differences between traditionally compiled and interpreted languages. We believe this tool, with refinements and improvements, could serve as an educational tool for compiling multiple languages in an environment where parallel comparisons can be made locally. Support for multiple kernels within the same notebook could allow for easy comparisons within the same book¹. Furthermore, our results supply a platform to test the premise of our literature review, suggesting interactive computing and programming is essential in productive programming. Future research could then be focused on a quantitative assessment of learning objectives using the JupyterHub environment.

¹ This is already possible, but the user must manually change the kernel any time they wish to use a different language.

References

- Blake, J. D. (2011). *Language considerations in the first-year CS curriculum*.
- Fangohr, H. (2004). A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering. In M. Bubak, G. D. van Albada, P. M. A. Sloot, & J. Dongarra (Eds.), *Computational Science—ICCS 2004* (Vol. 3039, pp. 1210–1217). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-25944-2_157
- Gehrig, R., & Martres, G. (n.d.). *Integration of Dotty in Jupyter notebooks*. 10. *Installing on Amazon Web Services—The Littlest JupyterHub v0.1 documentation*. (n.d.). Retrieved October 12, 2021, from <https://tljh.jupyter.org/en/latest/install/amazon.html>
- QuantStack. (2019, December 25). *Interactive Workflows for C++ with Jupyter*. Medium. <https://blog.jupyter.org/interactive-workflows-for-c-with-jupyter-fe9b54227d92>
- Singh, S. K. (2012). Performance Evaluation of Hybrid Reconfigurable Computing Architecture over Symmetrical FPGA. *International Journal of Embedded Systems and Applications*, 2(3), 107–116. <https://doi.org/10.5121/ijesa.2012.2312>
- Stroustrup, B. (2020). Thriving in a crowded and changing world: C++ 2006–2013;2020. *Proceedings of the ACM on Programming Languages*, 4(HOPL), 70:1-70:168. <https://doi.org/10.1145/3386320>
- The Littlest JupyterHub—The Littlest JupyterHub v0.1 documentation*. (n.d.). Retrieved October 12, 2021, from <https://tljh.jupyter.org/en/latest/>