Enabling Cross-Language Data Integration and Scalable Analytics in Decentralized Finance

Conor Flynn

Electrical and Computer Systems Engineering Rensselaer Polytechnic Institute Troy, United States of America flynnc3@rpi.edu

Aaron Green Mathematical Sciences Rensselaer Polytechnic Institute Troy, United States of America greena12@rpi.edu Kristin P. Bennett Mathematical and Computer Sciences Rensselaer Polytechnic Institute Troy, United States of America bennek@rpi.edu John S. Erickson Future of Computing Institute Rensselaer Polytechnic Institute Troy, United States of America erickj4@rpi.edu

Oshani Seneviratne Computer Science Rensselaer Polytechnic Institute Troy, United States of America senevo@rpi.edu

Abstract-With the agile development process of most academic and corporate entities, designing a robust computational back-end system that can support their ever-changing data needs is a constantly evolving challenge. We propose the implementation of a data and language-agnostic system design that handles different data schemes and sources while subsequently providing researchers and developers a way to connect to it that is supported by a vast majority of programming languages. To validate the efficacy of a system with this proposed architecture, we integrate various data sources throughout the decentralized finance (DeFi) space, specifically from DeFi lending protocols, retrieving tens of millions of data points to perform analytics through this system. We then access and process the retrieved data through several different programming languages (R-Lang, Python, and Java). Finally, we analyze the performance of the proposed architecture in relation to other high-performance systems and explore how this system performs under a high computational load.

Index Terms—Sockets, Database management, Multi-user channels

I. INTRODUCTION

Designing a robust computational back-end system poses a series of challenges many software engineers face daily. Languages all offer a variety of functionality, each with its benefits and drawbacks, causing developers to select a primary language based on the specifications of the project [1]. However, the selection of this language is often pivotal, as it determines not only the set of tools at a developer's disposal but often the ability to scale a system outside of its original confines.

Applications revolving around the transferring, storing, and processing of big data have been evermore prevalent as well. Although the retrieval of the data may be standardized to a certain degree through means such as OpenAPI standardization and Javascript Object Notation (JSON) formatting, the data being transmitted is not [2] [3]. Often this passed data has nested values (in the case of JSON, such as having JSON objects and JSON arrays), different primitive object types, and varying subsequent request mannerisms, all of which account for increased system complexity and the potential for overengineering solutions [4] [5] [6] [7].

From this, we define a problem that exists in both educational and business domains, being the lack of heterogeneity in data sets as well as the inability to expand upon a system in a practical manner. In this paper, we discuss the implementation and deployment of a system that accounts for these discrepancies between data sources and languages, resulting in a high-performance back-end solution for standardizing data transmission and storage between multiple data sources and user applications. We also analyze the underlying architecture of the system and how it can be independently utilized to create scalable solutions, how it is used in our application, and what it means for the system's architecture moving forward.

Based on these claims, we propose a solution called the "Decentralized Finance Data Engine," (DFDE) which utilizes the proposed system architecture. Decentralized Finance (DeFi) is a "new breed of consumer-facing financial applications composed as smart contracts, deployed on permissionless blockchain technologies" [8]. The architecture includes features such as multi-tenant architecture with asynchronous actions, programming language agnostic connections, and support for multiple data sources within the DeFi space focusing on different lending protocols such as the AAVE protocol [9] [10]. This implementation is then used in an educational lab setting, where researchers can easily pull and manipulate this data to expedite their work.

To introduce this work, we first cover related works with similar purposes followed by the relevancy of such a system and why it is beneficial to these different entities. Following this, we introduce the underlying architecture the DFDE is based on and analyze the performance of the design. Next we review proposed solutions for the internal and external connections such an implementation requires. Finally, we review the DFDE's architecture as a whole, how the different processes within it interact, and the applications it is used for.

II. RELATED WORK

Although this system proposes several improvements to the interoperability of systems and their transfer of data both internally and externally, one should recognize the existing developments in this area; primarily concerning the Application Programming Interface (API) specifications/protocols [11].

When regarding APIs and the manners to transfer data, almost all designs will fall under a specific API specification using either Remote Procedure Call (RPC), Service Object Access Protocol (SOAP), Representational State Transfer (REST), gRPC, or GraphQL. In this paper, we explore implementations using REST and GraphQL protocols.

A REST API refers to an API that follows six architectural constraints: uniform interface, stateless, cacheable, clientserver, layered system, and code on demand [12]. REST APIs are commonly used for create, read, update, and delete (CRUD) operations on remote systems, which is why they are primarily used for exposing server data to external clients [13].

GraphQL acts as a query language for APIs [11]. It allows users to specify the exact data needed from the source and then retrieve it in one call to the API. This approach aims to save both time and computational resources for the client and server and is adopted in scenarios requiring large amounts of data to be transferred.

RapidAPI [14] aims to generalize API calls through the use of a query language, similar to that of SQL and GraphQL, for the easy retrieval of data from an external source [15]. Similar to the proposed implementation, RapidAPI allows for MongoDB integration and connection as well as the caching of previously called data. Although very similar in usage to the proposed sample application, referenced in section IX-A, there are some key differences.

The primary difference is how the system communicates with end users. RapidAPI features a query language for users to specify explicit instructions which their system interprets [16]. Although great for the necessary applications, developers cannot easily expand the system or use it for other capabilities, which are extrapolated upon in the Routing Architecture section. Its usage also revolves around a query language, which may not be language agnostic as using something lowlevel such as Sockets [17]. Finally, their requesting system is much more specific than the proposed implementation. This has benefits, such as being able to request data in a more specific manner, while also having its limitations, such as more specifications being required when executing a call. It cannot also perform recursive API calls automatically, requiring that handling be done on the user's end.

III. RELEVANCY

Before analyzing the underlying architecture of the DFDE and its processes, one must understand the relevancy of such a system and the problems to which it can be applied. Primarily, the DFDE aims to resolve problems in relation to API design; however, these questions can be expanded upon to any form of data transfer. As Lindman states regarding business, architecture, process, and organization (BAPO) perspectives:

The main concern of this perspective is to investigate the technical issues associated with API design and development, similar to the API layer presented but from a broader perspective (architecture of the interacting systems as opposed to API design). Questions to ask include: How does one manage API versioning (e.g., side-by-side deployment of different versions)? How does one design APIs for an extension? How does one check the backward compatibility of APIs between different versions? [18]

This gives insight into the troubles faced by researchers regarding designing APIs (or any outward-facing programs). However, the DFDE offers a solution to many of these posed questions.

Firstly regarding versioning and backward compatibility, the DFDE does inherently hold backward compatibility functionalities so long as the underlying Router structure and connection mechanisms are not modified. Processes may change. However, unlike APIs, where the structuring of data, endpoint connection, and parameters may change, the DFDE's internal calling to external APIs will always remain consistent. The only inconsistency that may be present is an external data provider's API connection may change, causing the DFDE's internal storage of data and the outward-facing connection to be incorrect, requiring manual maintenance by the user of the DFDE to fix the connection. This makes it so researchers have to make minimal changes to their existing applications using the DFDE, allowing for minimal obstruction to the lab's workflow with any sudden pivots in the focal research topic.

Similarly, extensions to the DFDE can be made easy thanks to the Router design. With the only internal modification requirement being connecting the new Router(s) to the rest of the system, there is not much else a programmer has to do to expand upon the DFDE. To access the new Router(s) and sub-tag(s), users simply have to specify their location when submitting requests. All internal processes are modular, making future updates easy. With this, any students working on a project irrespective of being an original developer can easily add their own components to the DFDE.

Finally, due to the data-agnostic designs of the DFDE, new data sets can be easily parsed, stored, and analyzed by researchers. The language-agnostic design also allows students to work in their preferred languages, increasing their productivity and the efficacy of their work while helping to reduce the friction in learning a new language [19]. This allows for the usage of such a system in a classroom setting as is currently being done with the proposed application of the DFDE, allowing students to explore material outside of the confines of the course.

IV. ROUTING ARCHITECTURE

This section describes the internal data routing architecture of the DFDE, on which all processes are inherited.

A. Overview

To design a language and data-agnostic system, a highly versatile and robust architecture was needed. Therefore the proposed architecture is modeled after a network consisting of Routers and Packets, with different Packets of data being able to be transmitted freely between the connected Routers [20]. We use fig. 1 to give a visual of the routing architecture.

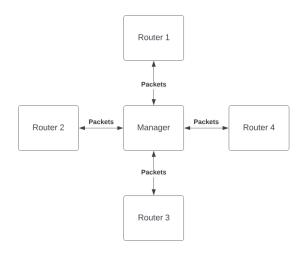


Fig. 1. Routing Architecture

B. Managers

Managers are the primary link between all Routers, with each Router sharing the same Manager. All Routers within a manager are stored using a HashMap, with the key being the unique "Tag" of a Router and the value being the Router object itself [21]. This way, there is a fast connection when two Routers are trying to communicate, as a HashMap uses constant time when accessing information [22].

Managers also are "intelligent" objects recognizing when two groups of Routers, each with a different Manager, connect. In such cases, they will override all of the Routers to have the same Manager so that constant run-time is retained. Managers also handle all edge case connections, such as duplicate connections, disconnections, and threaded processes [23].

C. Routers

Routers are the interactive endpoints for all processes. They contain standardized functions that allow for easy connection and communication with other Routers. Each Router has a "Tag," which is a unique String identifier so a Manager can recognize it and properly route Packets to it [24]. Routers also consist of processes, all denoted by a unique String called a "Sub-Tag" for proper Packet transit within the Router. Each process is a defined method that returns a Response containing specific information regarding the call [25].

Similar to Managers, Routers also contain "intelligent" capabilities such as the ability to: recognize specifically formatted methods as process functions (with unique identifiers), send Packets to these requested functions, and recognize any formatting errors when defining these functions during the initial run-time compilation. To accomplish some of these functionalities, the architecture uses a feature of the Java language called Reflection, which allows for the Java program to analyze the code on which it is running and perform actions accordingly [26], [27].

D. Packets

Packets are lightweight objects comprised of four main pieces of information: the sender, the receiver (referred to as the "Tag"), the process (referred to as the "Sub-Tag"), and all internal data [28]. The sender, Tag, and Sub-Tag are all listed as Strings, whereas the data is stored as a HashMap (with both the key and values being Strings) [24] [21].

Each call to a Router's process then returns a Response. Responses contain information regarding the call's outcome and are comprised of a response code, a message, and optional return data. The response code is a predefined integer representing a specific outcome (commonly following the OpenAPI HTTP Status Code standard) [6], [29]. The message is also predefined and is a String corresponding to the returned response code, giving more insight into the response of the code. Finally, the data refers to any optional data requested by the Packet that may need to be returned. Although the data is returned as a String, it can be cast into any needed data type by the requesting process [30].

E. Review

When using this architecture in the implementation and design of a large system, several benefits and drawbacks can be found, with their severity on the implementation varying based on the requirements.

Benefits include the easy scalability of the system, the generalized data flow, and the performance, later discussed in the section V section. When designing new systems, developers may often specify the system to the current needs, not foreseeing future updates and improvements. This often creates a very rigid programming schema, requiring developers to spend more time when making these updates. The architecture proposed in this paper solves this problem, allowing for the easy implementation and addition of new components without requiring explicit modification to existing regions. Similarly, this generalized data flow allows for easy communication between different sections of the system, making any changes easy to implement. Added latency is also very minimal, making it a very strong contender on time-sensitive programs.

The primary drawback of this architecture is the generalized data flow, specifically the casting required to transmit information. Since all data is passed as a string, it may often be challenging to communicate complex objects between portions of the system. A solution to this could be to create the Packets using an Object value rather than a String. However, further checks would need to be done to ensure the safety of casting the object [31].

V. PERFORMANCE ANALYSIS

This section discusses the performance of the proposed routing architecture and how one should interpret the results.

A. Experimental Setup

Hardware: The environment used contains a single Intel(R) i9-12900KF processor with 16 cores, in which each core runs at 3.187 GHz and 64GB of memory.

Software: The environment is using Microsoft Windows 10 Pro version 10.0.19045 Build 19045 as an operating system. The programming language Java is used to run the tests, using Java 20.0.1, build 20.0.1+9-29.

B. Overview

To analyze the performance of the proposed architecture, we test the rate at which two Routers can submit packets of data to one another. To accomplish this, we send a packet containing some information to a sample protocol on a connected Router. We then analyze the time it takes for that packet to reach the Router and for a response to be received. Once received, we send another packet and repeat the cycle for 10 seconds until the run is up. We then average the packets sent per second over these 10 seconds to determine the performance.

We also analyze the maximum capacity of the architecture through threading, calculating how many packets can be sent asynchronously. To understand how threading affects performance, we begin with a singularly threaded test and increase the number of threads by a factor of 10 until we reach 90 concurrent threads.

C. Analysis

We first review the performance of each thread regarding the number of packets being sent. As seen by fig. 2. There is a significant drop in performance between 1 and 10 threads, which is to be expected. However, this degradation in performance becomes more minimal as more threads are used. From this, we can determine the time it takes for a packet to be sent and received by dividing the number of nanoseconds per second (1 billion) by the number of packets sent per second. This tells us that it takes 218.78ns to send a packet within a single thread versus 4835.81ns to send a packet when 90 threads are running concurrently.

Although this drop in performance is very large, we can also extrapolate the total number of packets being sent per second. To do this, we take the current number of packets sent per second and multiply it by the number of threads actively running. As seen in fig. 3, although the speed at which packets are sent is greatly reduced, the number of overall packets being sent is greatly increased, with 90 threads handling over 4 times the number of packets as a single thread.

From this, we can see the trade-off of performance versus capacity. Should a system need fast interactions between processes, it may opt to limit the number of threads so that performance is retained. Likewise, if an application relies on transmitting large quantities of information with minimal emphasis on time, it may choose to operate with a large number of threads.

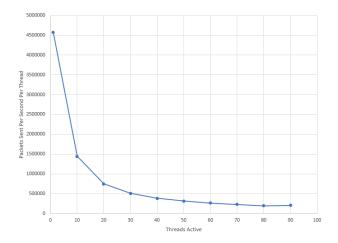


Fig. 2. Packets Sent Per Second Per Thread

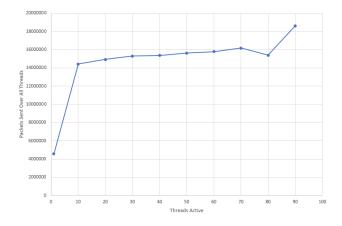


Fig. 3. Packets Sent Per Second Across All Active Threads

VI. INTERNAL STORAGE

This section covers all internal architecture relating to the processing, storage, and retrieval of all agnostic data handled by the DFDE.

A. Data Schema

Standardizing a database with no context of the incoming data is certainly challenging. To solve this problem, we rely on the preliminary condition of the data being structured properly before the DFDE interprets it. The required format is JSON, a lightweight, text-based, language-independent data interchange format [3]. From this, it can be easily converted into Binary JSON (BSON) so the MongoDB can properly process the data points [32]. Through the use of recursion, we can easily extract information embedded within the calls and use it if necessary [33]. Use cases include recursive calls to endpoints, extracting storage information, and fast data filtering.

B. Collection Management

To house the formatted data points, MongoDB relies on Collections [34]. Collections are groups of data points stored under a common name, which a database can easily access. This includes two primary components: the request name and date (should the data be dated). Request names are required to be unique on initialization, with the DFDE checking each request for uniqueness. This unique name allows for consistency within the DFDE, allowing for calls to be validated if the data has already been received or not. For dated data, the naming schema changes slightly. Rather than just being the unique name passed, it will also include the date to be stored in the format: name-yyyy-MM-dd with the year, month, and day following the standardized "Formatting and parsing date-time patterns" [35]. Although this schema is trivial, it allows for a variety of functionality which will be further expanded upon in later sections.

C. Review

There are many benefits to using a data-agnostic system such as the DFDE, as described above, which include easy accessibility, reduction of latency, and credential obfuscation. Easy accessibility refers to the availability of the data to both users operating with the external protocols mentioned later as well as to users directly interacting with the MongoDB [36]. Reduced latency refers to the reduction of calls to external data sources since called data is cached and stored for subsequent calls. Finally, credential obfuscation means that one user can request data using an API key and subsequent calls to retrieve the data without requiring the key, as the data is already cached and stored properly.

VII. EXTERNAL DATA CONNECTIONS

This section covers all external connections to data sources outside the scope of the DFDE. This includes but is not limited to, external REST APIs, WebSockets, and Web3 connections [37]–[39].

A. Abstract Connections

Designing a system to account for every existing type of available endpoint is nearly impossible. Between different endpoint schemes used by companies, the types of connections offered (such as REST APIs and WebSockets), and the data returned by these endpoints, a system would have to be needlessly over-engineered, defeating the purpose of a highfrequency data transferring system.

To counter this issue, we implemented generic and commonly used protocols while also leaving the DFDE open to abstraction for further development. This way, should the integrated solutions not be suited for a given project, developers can easily integrate their own abstract endpoints with the required specifications.

B. Configuration

Before designing a system to cater to these different data sources, we first need to comprise a way for the DFDE to easily interpret new endpoints. This can be achieved through the usage of properties files, which are files that contain specific key-value pairs able to be read by the DFDE. By creating a directory that houses these different configuration files, the DFDE can load them on initialization and recognize/handle incoming requests.

C. REST APIs

The primary generalized architecture supports external calls to REST APIs [37]. REST APIs consist of several different properties, primarily: a Uniform Resource Locator (URL), properties defined after the URL, and headers to specify specific underlying properties not passed within the URL [40]. There are also edge cases, such as URLs having a unique path differing from the one specified, which the DFDE must also account for. To handle these different cases and information required, we have variables within the properties files such as url.base,url.properties, and url.headers which will properly translate the passed requests into the proper REST API calls necessary to retrieve information.

Other information can be necessary, such as retrieving dated data. Rather than having users make individual calls to retrieve each date, we can use a recursive function to extrapolate every date and make each call on behalf of the user. For this, we have key-value pairs that define the variables used for setting the properties controlling the dates, such as date.start and date.end as well as ones defining how the date is formatted date.format using the same schema as the collection management system [35].

Finally, the DFDE has to account for recursive calls. This is because most external systems have a limit as to how many data points a single REST API call can return. To determine if a recursive call has reached the end of the data (stored within the given dates or endpoint), we use a limiter. If the number of data points returned is below the limit defined, we deem the call finished allowing it to move to the next call. While designing the DFDE, we identified four main types of recursive calls: Single, URL, Incremental, and Static.

1) Single: A single call is the most basic type of call. It tells the DFDE that no recursion is required and there is only one request needed to retrieve all the data. This is commonly used for retrieving static data sets containing reference data.

2) URL: A URL call states that there is an iterative URL returned with the data that the user should use to make the next call. For this type of call, the DFDE just needs to know where the new URL is located in the returned data, and it will be able to substitute it to make the next call.

3) Incremental: An Incremental call tells the DFDE that there is a variable that needs to be increased to retrieve the next group of data. Commonly used for endpoints that have "Pages" of data, and the page needs to be increased by one each call to obtain the next set of data.

4) Static: A static call refers to a data point that is returned inside the passed call that needs to replace a parameter in the URL. This is typically a timestamp or index where by passing the timestamp or index, the external data source will return a certain number of data points after that value. Due to the probability of more niche cases than these, the DFDE allows for the abstract configuration of external calls, as mentioned previously.

D. WebSockets

WebSockets refer to a form of continuous communication and data flow, where one system acts as a client and the other as a server [38]. This form of constant communication is seen primarily in live data feed applications. To integrate connections such as these into the DFDE, all that is necessary is an establishment of the connection and the usage of the PUSH call as defined by the LocalStreamHandler.

E. Web3

Designing a Web3 connection is most likely the most specialized type of connection supported by the DFDE, as there are many ways to connect to decentralized applications. Later in this paper, we will explore connections to AAVE and The Graph, both of which are Web3 applications. However, should users want to obtain other Web3 data, it will most likely require them to develop a new connection utilizing the abstract connection types discussed previously [41], [42].

VIII. LANGUAGE AGNOSTIC CONNECTIONS

This section contains information regarding the programming language features required for connection, the steps to connect, and the embedded multi-tenant architecture of the DFDE [9].

A. Language Feature Requirements

Outside of common object-oriented programming functionalities, the only language requirement is the support of Socket connections. [43] [44]. Since Sockets are considered a basic programming feature, most modern generic programming languages will support their usage [17].

Sockets were selected as the primary method of choice for transmitting data for a few reasons. Commonly referred to as WebSockets, this form of communication "provides a full-duplex, bidirectional communication channel that operates through a single Socket over the Web" [45]. This gives them several large advantages over REST API calls with functionalities such as reading and writing capabilities, noncapped data transfer (typically capped at 2MB for common server instances), and a constant line of communication [46]. The usage of all of these functions will be explored in later sections.

B. Initializing Connections

There are a few steps required to connect to the DFDE before requesting and retrieving data.

1) Destination Key Retrieval: Upon a user initiating a connection with the DFDE, the DFDE will submit a one-line response to the Socket containing a unique key. This key, referred to as the destination-key, is the unique identifier of the application. It tells the DFDE where to send data that is requested. When a user requests data from the DFDE, they are also required to submit their destination-key so that the DFDE can properly transmit their data. One benefit of this is an application can asynchronously request data to be sent to other applications by sharing each other's keys.

2) Sending a Request: Due to the way requests are processed, the DFDE has strict syntactical requirements for the formatting of requests. Since the DFDE utilizes the Router architecture previously discussed, a request can be made to any Router's tag and sub-tag. To send requests to the DFDE, we utilize the tag SRC (the stream registry controller) and the sub-tag RQST (the request protocol).

Next, we specify the required details relating to the call. To handle these different parameters, the DFDE has a specified delimiter (defaulted to &&&) which is used to isolate these values. For calls to RQST, we have the required value of protocol which specifies which external call you want to make. For example, this could be defined as graph-aave-users which tells the DFDE to retrieve user data from AAVE using The Graph [10] [42].

Finally, we pass optional parameters which may be specific to the call. These typically fall under the values: properties, headers, start_date, and end_date. Both properties and headers refer to the standard URL parameters passed through a common REST API call [47]. The date parameters start_date and end_date are used to specify the dates of the data should it be a dated call.

A sample call to the DFDE would be formatted as follows (note with no line breaks):

```
SRC&&&RQST&&&protocol&&&graph-aave-users
&&&start_date&&&2022-01-01
&&&end_date&&&2023-01-01
```

This would send a request to graph-aave-users to retrieve data all user data for the year 2022.

3) Parsing the Response: Finally, the user application has to properly handle all incoming data being sent through the Socket. The format for all data passed is in JSON, with each data point being sent as an individual JSON object rather than in an array [32] [48]. This can be easily processed in most languages and then stored however the user application needs.

The user application also needs to know when the data flow ends, as the connection will not automatically disconnect. Therefore, once the DFDE is done transmitting data, it will pass the character sequence <<<end>>> signaling the end of the data transmission.

To keep the connection alive on larger data set requests, as Socket connections will automatically timeout after a short period of time, we use a "heartbeat" connection [49] [50]. This is passed to the user at every predetermined interval of time (defaulted to 5 seconds) as the character sequence <<<<heartbeat>>>. If the user receives this line, they can

just ignore it and continue to the next line, as it is not part of the transmitted data.

C. Multi-tenant Architecture

Multi-tenant architecture refers to the usage of the same computational resources by multiple external users. In the instance of the DFDE, we utilize multi-tenant architecture by hosting a threaded ServerSocket application for users to connect to [51] [52]. fig. 4 gives a visual of how the OutputHandler Router within the DFDE acts as a Server-Socket being able to handle multiple Socket connections at once.

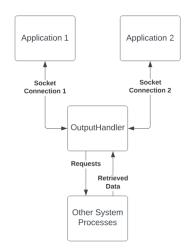


Fig. 4. Socket Connection Overview

To keep track of these various Socket connections, we use the destination-key mentioned previously. The OutputHandler keeps a registry of each of these Socket and key pairs, routing all of the connections and requests for the DFDE.

The handler also splits the storage of the consumer (incoming data stream) and producer (outgoing data stream), allowing each of them to work asynchronously. This is because only the consumer, the channel used for receiving user requests, is bound to a thread, allowing for the outward data stream to be accessed by multiple threads.

IX. IMPLEMENTED APPLICATION

In this section, we will discuss the currently implemented application of the proposed architecture, the DFDE, and how it is working to improve the efficiency and quality of data analytics performed in a lab setting. This particular application is for the analysis of transaction data from DeFi protocols. The size and structure of the data for this application are described in section IX-C.

A. DeFi Data Engine

To demonstrate the efficacy of the designs discussed, we present the full architectural implementation of the DFDE. This engine utilized the Router system, internal storage mechanism, external agnostic data retrieval system, and Socket connections for communications. By using all of these systems, we can create a flow chart for the different Router connections and their interactions, as shown in fig. 5. Each of the processes listed here is a Router, all of which contain multiple sub-processes, with the lines showing the Routers' requests between each other.

Several of these implemented Routers are imperative to the functionality of the engine. For managing data streams, the engine relies on the LocalStreamHandler (LSH), the ExternalStreamHandler (ESH), and the StreamRegistryController (SRC) to handle and process requests. When handling events such as user connections and data flows, the engine utilizes the OutputHandler (OUT). And finally, for internal communication and engine stability/information, it utilizes the Engine (ENG) and the Logger (LOG) Routers, with all other defined Routers supporting infrastructure-related processes of the system. From these brief descriptions, as well as through reference to fig. 5, we can get a better understanding of how different data streams and requests flow throughout the system.

The most common request is the request for stored data, where the system will either retrieve it from the historical database or from an external source. In fig. 6, we show a simplified version of the flow throughout the different Routers to execute a request.

B. Sample Connections

To demonstrate the language-agnostic connection capabilities of the DFDE, sample connections were developed in R-Lang [53], Python [54], and Java [55]. These examples utilize the aforementioned parameters required to send a request (protocol, properties, headers, startdate, and enddate) and automatically instantiate the Socket connection with the server the engine is hosted on. After interpreting the request and receiving the ending statement (<<<end>>>), these systems will terminate the connection with the engine and output the returned data. The sample implementations for R-Lang and Python can be found on Github.

C. Data Retrieval

To further expand upon the functionality of the engine, research was conducted using it within a laboratory setting, with the primary objective of the research being the analysis of data from DeFi lending protocols as illustrated in these papers [56], [57]. Using the DFDE for these analyses showcases the engine's ability to retrieve large datasets. The data acquired for these analyses included more than 60 million transactions from various DeFi lending protocols such as Aave [58], [59], Compound [60], and MakerDAO [61]. Retrieving this data required the engine to connect to dozens of different tables from The Graph [42], API endpoints from Amberdata [62], and API endpoints from DeFi Llama [63]. From The Graph, GraphQL [64] was used to acquire raw transaction data for seven different markets of Aave. From Amberdata, a REST API connection [2] was used to acquire raw transaction data for Compound, MakerDAO, and one additional Aave market;

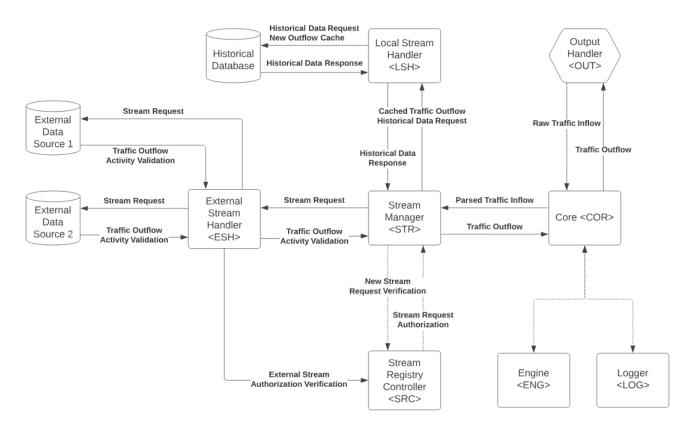


Fig. 5. DeFi Data Engine Architecture

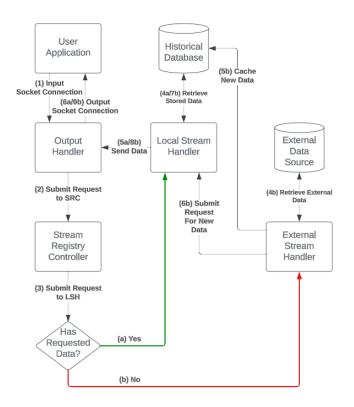


Fig. 6. Sample Request Flow

blockchain address data was also acquired from Amberdata. From DeFi Llama, we pulled data for classifying cryptocurrencies as stable or non-stable coins using a REST API. Retrieving data from multiple sources aims to showcase the data-agnostic design of the engine.

Aside from the retrieval of the data, preliminary processing is required to get the data in the form the researchers require. For this, the sample file of GetTransactions.Rmd is created, which handles all internal processing of the data so that it can be loaded easily with a simple function call [65]. It makes use of 13 sample requests, showcasing the data-agnostic designs of the engine [66].

Once the data is all retrieved and processed, it will be presented to the user in a list of two objects. The first object within the list is response which details the outcome of the call to the engine. This contains the code (code corresponding to the outcome of the call), message (accompanying message to the response code), and data (any data that may be returned from the call). The second object is the data frame previously mentioned under the key df, which contains all returned and processed data.

D. Data Analytics

Two sample plots are used to show how the retrieved processed data can be used in research. Each only uses data retrieved from the engine with no external supporting data.

The first plot, shown in fig. 7, is a k-means clustering plot assigning different clusters to users of the AAVE lending

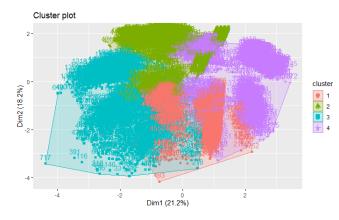


Fig. 7. User Clustering Plot

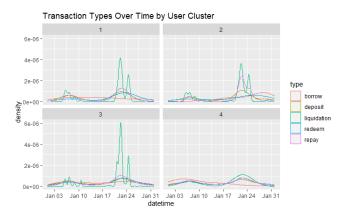


Fig. 8. Transaction Type Density Plot

protocol based on their skill level and risk profile derived from their transaction history [56], [67]. From this chart, we can see this clustering and each of the different users found within the lending transaction data being placed into one of four categories.

The second plot, shown in fig. 8, is a density plot, which informs the researcher how many of each transaction type occurred during a certain time period based on the clusters found in fig. 7.

Although these plots are demonstrative visual overviews of the data retrieved, more meaningful points can certainly be extrapolated with more time based on the given researchers' needs.

X. CONCLUSION

To further improve this application of the proposed systems, developers and researchers can make use of further external communication generalizations. Outside of the previously discussed REST API generalized connections already implemented, developers can potentially integrate connections such as live connections to external Websocket applications, generalized connections to other query languages such as SQL and NoSQL databases, and further improve the efficiency of the internal designs [15]. Consequently, we have proposed a language and data agnostic system called the "DeFi Data Engine" which aims to generalize data retrieval and storage from multiple external sources for easy usage. First, we explored the internal architecture of the system, analyzing its performance capabilities, how it can be used to scale systems, and the embedded data agnostic communication architecture. Then, we discussed the internal and external communication processes developed to support the DFDE and how the implementation can handle the different formatting of data from these sources. Finally we discuss the DFDE being used as an application in classroom and lab settings, as well as the results and papers that were derived from its usage.

XI. ACKNOWLEDGEMENTS

The authors acknowledge the support from NSF IUCRC CRAFT center research grants (CRAFT Grants #22003, #22006) for this research. The opinions expressed in this publication and its accompanying code base do not necessarily represent the views of NSF IUCRC CRAFT. This work was supported by the Rensselaer Institute for Data Exploration and Applications (IDEA).¹ We also thank Amberdata for supporting us with access to their data.

REFERENCES

- L. Goosen, E. Mentz, and H. Nieuwoudt, "Choosing the "best" programming language," in *Proceedings of the computer science and IT* education conference. Citeseer, 2007, pp. 269–282.
- [2] C. Mulligan, "Open api standardisation for the ngn platform," in 2008 First ITU-T Kaleidoscope Academic Conference - Innovations in NGN: Future Network and Services, 2008, pp. 25–32.
- [3] T. Bray, "The javascript object notation (json) data interchange format," Tech. Rep., 2014.
- [4] "Jsonobject." [Online]. Available: https://docs.oracle.com/javaee/7/api/ javax/json/JsonObject.html
- [5] "Jsonarray." [Online]. Available: https://docs.oracle.com/javaee/7/api/ javax/json/JsonArray.html
- [6] "Primitive data types." [Online]. Available: https://docs.oracle.com/ javase/tutorial/java/nutsandbolts/datatypes.html
- [7] "What is overengineering?" Jan 2008. [Online]. Available: https: //www.codesimplicity.com/post/what-is-overengineering/
- [8] J. R. Jensen, V. von Wachter, and O. Ross, "An introduction to decentralized finance (defi)," *Complex Systems Informatics and Modeling Quarterly*, no. 26, pp. 46–54, 2021.
- [9] F. Chong, G. Carraro, and R. Wolter, "Multi-tenant data architecture," MSDN Library, Microsoft Corporation, pp. 14–30, 2006.
- [10] "Aave open source liquidity protocol." [Online]. Available: https: //aave.com/
- [11] "What is an api: Definition, types, specifications, documentation," Nov 2022. [Online]. Available: https://www.altexsoft.com/blog/engineering/ what-is-api-definition-types-specifications-documentation/
- [12] R. T. Fielding, Architectural styles and the design of network-based software architectures. University of California, Irvine, 2000.
- [13] "Rpc vs rest difference between api architectures aws," Aug 2023. [Online]. Available: https://aws.amazon.com/compare/ the-difference-between-rpc-and-rest/
- [14] "What is rapidql?" [Online]. Available: https://docs.rapidql.com/docs
- [15] M. Stonebraker, "Sql databases v. nosql databases," Communications of the ACM, vol. 53, no. 4, pp. 10–11, 2010.
- [16] "Rapidql." [Online]. Available: https://github.com/RapidAPI/rapidql
- [17] L. Kalita, "Socket programming," International Journal of Computer Science and Information Technologies, vol. 5, no. 3, pp. 4802–4807, 2014.

¹https://idea.rpi.edu/

- [18] J. Lindman, J. Horkoff, I. Hammouda, and E. Knauss, "Emerging perspectives of application programming interface strategy: A framework to respond to business concerns," *IEEE Software*, vol. 37, no. 2, p. 55, 2020.
- [19] N. Shrestha, C. Botta, T. Barik, and C. Parnin, "Here we go again: Why is it difficult for developers to learn another programming language?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 691–701.
- [20] "How routers transfer packets planning for network deployment in oracle solaris 11.2," Nov 2014. [Online]. Available: https: //docs.oracle.com/cd/E36784_01/html/E37474/ipplan-43.html
- [21] "Hashmap (java platform se 8)." [Online]. Available: https://docs.oracle. com/javase/8/docs/api/java/util/HashMap.html
- [22] D. Bajracharya and N. Kathmandu, "A review on java hashmap and treemap."
- [23] A. I. Holub and A. Hollub, Taming Java Threads. Springer, 2000.
- [24] "Strings." [Online]. Available: https://docs.oracle.com/javase/tutorial/ java/data/strings.html
- [25] "Defining methods." [Online]. Available: https://docs.oracle.com/javase/ tutorial/java/javaOO/methods.html
- [26] I. R. Forman, Java reflection in action, 2005.
- [27] ronmamo, "Reflections," Oct 2021. [Online]. Available: https://github. com/ronmamo/reflections
- [28] "Defi data engine wiki architecture packets," May 2023. [Online]. Available: https://github.rpi.edu/DataINCITE/IDEA-DeFi-CRAFT/wiki/ Architecture
- [29] "Openapi specification v3.1.0," Feb 2021. [Online]. Available: https: //spec.openapis.org/oas/v3.1.0
- [30] May 2021. [Online]. Available: https://www.baeldung.com/ java-type-casting
- [31] A. Fruit and S. Rectangle, "Object-oriented programming," 1993.
- [32] [Online]. Available: https://www.mongodb.com/json-and-bson
- [33] E. W. Dijkstra, "Recursive programming," in Edsger Wybe Dijkstra: His Life, Work, and Legacy, 2022, pp. 291–300.
- [34] [Online]. Available: https://www.mongodb.com/docs/manual/core/ databases-and-collections/
- [35] 03 2021. [Online]. Available: https://www.ibm.com/docs/en/rtw/9.1.1? topic=reference-formatting-parsing-date-time-patterns
- [36] "Mongodb documentation." [Online]. Available: https://www.mongodb. com/docs/
- [37] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, and G. Percannella, "Rest apis: A large-scale analysis of compliance with principles and best practices," in *Web Engineering: 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings 16.* Springer, 2016, pp. 21–39.
- [38] I. Fette and A. Melnikov, "The websocket protocol," Tech. Rep., 2011.
- [39] S. Yang and M. Li, "Web3. 0 data infrastructure: Challenges and opportunities," *IEEE Network*, vol. 37, no. 1, pp. 4–5, 2023.
- [40] T. Berners-Lee, L. Masinter, and M. McCahill, "Uniform resource locators (url)," Tech. Rep., 1994.
- [41] E. Frangella and L. Herskind, Aave V3 Technical Paper, Jan 2022.
- [42] [Online]. Available: https://thegraph.com/
- [43] B. Stroustrup, "What is object-oriented programming?" *IEEE software*, vol. 5, no. 3, pp. 10–20, 1988.
- [44] M. J. Donahoo and K. L. Calvert, TCP/IP sockets in C: practical guide for programmers. Morgan Kaufmann, 2009.
- [45] V. Pimentel and B. G. Nickerson, "Communicating and displaying realtime data with websocket," *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, 2012.
- [46] "Apache tomcat configuration reference the http connector." [Online]. Available: https://tomcat.apache.org/tomcat-5.5-doc/config/http.html
- [47] "Http header fields and uri parameters." [Online]. Available: https://www.ibm.com/docs/en/odm/8.10?topic= api-http-header-fields-uri-parameters
- [48] "What is json, json object and json array?" [Online]. Available: https://toolsqa.com/rest-assured/what-is-json/
- [49] "Timeouts." [Online]. Available: https://websockets.readthedocs.io/en/ stable/topics/timeouts.html
- [50] "Ibm documentation." [Online]. Available: https://www.ibm.com/docs/en/powerha-aix/7.2?topic= heartbeating-over-tcpip-storage-area-networks
- [51] "Serversocket." [Online]. Available: https://docs.oracle.com/javase/8/ docs/api/java/net/ServerSocket.html

- [52] S. Kleiman, D. Shah, and B. Smaalders, *Programming with threads*. Sun Soft Press Mountain View, 1996.
- [53] "R: The r project for statistical computing," Jun 2016. [Online]. Available: https://www.r-project.org/
- [54] G. Van Rossum and F. L. Drake Jr, *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995, vol. 620.
- [55] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language*. Addison Wesley Professional, 2005.
- [56] A. Green, C. Cammilleri, J. S. Erickson, O. Seneviratne, and K. P. Bennett, "Defi survival analysis: Insights into risks and user behaviors," in *The International Conference on Mathematical Research for Blockchain Economy*. Springer, 2022, pp. 127–141.
- [57] A. Green, M. Giannattasio, K. Wang, J. S. Erickson, O. Seneviratne, and K. P. Bennett, "Characterizing common quarterly behaviors in defi lending protocols," 2023.
- [58] Boado, Ernesto, "AAVE Protocol Whitepaper V2.0," Tech. Rep., 12 2020. [Online]. Available: https://cryptorating.eu/whitepapers/Aave/ aave-v2-whitepaper.pdf
- [59] Emilio Frangella, Lasse Herskind, "AAVE V3 Technical Paper," Tech. Rep., 01 2022. [Online]. Available: https://github.com/aave/ aave-v3-core/blob/master/techpaper/Aave_V3_Technical_Paper.pdf
- [60] Robert Leshner, Geoffrey Hayes, "Compound: The Money Market Protocol," Tech. Rep., 02 2019. [Online]. Available: https://compound. finance/documents/Compound.Whitepaper.pdf
- [61] MakerDAO, "The Maker Protocol: MakerDAO's Multi-Collateral Dai (MCD) System," Tech. Rep. [Online]. Available: https://makerdao.com/ en/whitepaper/#abstract
- [62] "Amberdata." [Online]. Available: https://www.amberdata.io
- [63] "Defillama." [Online]. Available: https://www.defillama.com
- [64] "Graphql a query language for your api." [Online]. Available: https://graphql.org/
- [65] C. Flynn, "Gettransactions.java." [Online]. Available: https://github.rpi.edu/DataINCITE/IDEA-DeFi-CRAFT/blob/ main/Materials\/All%20DeFi%20Data%20Engine%20Content/ R-Code-Samples/GetTransactions.Rmd
- [66] —, "Requests directory." [Online]. Available: https: //github.rpi.edu/DataINCITE/IDEA-DeFi-CRAFT/blob/main/Materials\ /All%20DeFi%20Data%20Engine%20Content/DeFi-Data-Engine/ DeFi%20Data%20Engine/src/main/resources/requests
- [67] A. Likas, N. Vlassis, and J. J. Verbeek, "The global k-means clustering algorithm," *Pattern recognition*, vol. 36, no. 2, pp. 451–461, 2003.