12-2023

Towards Expressive and Versatile Visualization-as-a-Service (VaaS)

Tanner C. Hobson
University of Tennessee, Knoxville, thobson2@vols.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Tanner C. Hobson entitled "Towards Expressive and Versatile Visualization-as-a-Service (VaaS)." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jian Huang, Major Professor

We have read this dissertation and recommend its acceptance:

Audris Mockus, Michela Taufer, Jitendra Kumar, Tom Peterka

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Towards Expressive and Versatile Visualization-as-a-Service (VaaS)

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Tanner Hobson

December 2023

Dedicated to Kelly, the love of my life

Acknowledgments

I am eternally grateful for all of the support I have received over the years. Without the influence of each of the following people, I would never have reached this conclusion of my academic career.

I am especially thankful for the guidance of my advisor, Dr. Jian Huang. Jian's patient mentorship and unwavering faith in my potential have pushed me to become my best self. When I felt the need to give up and quit, it was always Jian's steadfast belief in me that pushed me to strive for excellence. I owe a significant portion of my personal and professional growth to his invaluable guidance.

I am also incredibly fortunate to have married my favorite person and greatest supporter, Kelly. Her unwavering support has been an anchor in my life, sustaining me through the highs and lows of my academic career. Even when I had no faith in myself, I always knew I had Kelly's support. Her own journey through academia has been a constant source of inspiration and motivation.

I have had the opportunity to work with some of the best people I could hope for. I am especially thankful for the connections I have made with Moa Raji and Alok Hota. There were many days of insightful discussions within the Seelab office, paired with equally as many unproductive, but fun, conversations. Their outside perspectives and support have been instrumental in my growth, and for that, I am very thankful.

Finally, I would like to express my appreciation to my committee members Dr. Audris Mockus, Dr. Michela Taufer, Dr. Jitendra Kumar, and Dr. Tom Peterka for their collective expertise and continued support.

Abstract

The rapid growth of data in scientific visualization has posed significant challenges to the scalability and availability of interactive visualization tools. These challenges can be largely attributed to the limitations of traditional monolithic applications in handling large datasets and accommodating multiple users or devices. To address these issues, the Visualization-as-a-Service (VaaS) architecture has emerged as a promising solution. VaaS leverages cloud-based visualization capabilities to provide on-demand and cost-effective interactive visualization. Existing VaaS has been simplistic by design with focuses on task-parallelism with single-user-per-device tasks for predetermined visualizations. This dissertation aims to extend the capabilities of VaaS by exploring data-parallel visualization services with multi-device support and hypothesis-driven explorations. By incorporating stateful information and enabling dynamic computation, VaaS' performance and flexibility for various real-world applications is improved. This dissertation explores the history of monolithic and VaaS architectures, the design and implementations of 3 new VaaS applications, and a final exploration of the future of VaaS. This research contributes to the advancement of interactive scientific visualization, addressing the challenges posed by large datasets and remote collaboration scenarios.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

In the field of scientific visualization, the growth of data has far outpaced the growth of personal computing resources, which presents a major challenge to the data scalability of interactive visualization tools. This data challenge is deepening the barriers of reproducibility, sharability, and even basic accessibility of interactive scientific visualization. Traditional monolithic applications are poorly suited to handle large datasets interactively since they bind all aspects of visualization including data loading, rendering, and user interfaces rigidly into the same package. Additionally, monolithic applications often enforce a single-user-to-single-device scenario, limiting their potential to scale to multiple collaborating users, or the same user on multiple devices simultaneously. Despite many attempts to split up monolithic applications into a client-server model, there remains an open problem of how to provide interactive visualization of large data sets to users in remote, collaborative, and multi-device scenarios.

The Visualization-as-a-Service (VaaS) architecture has emerged as a potential solution to this open problem. VaaS situates visualization capabilities and services in the cloud, vastly improving on-demand availability of interactive visualization tools. An early example of VaaS is Tapestry, a high-performance volume rendering microservice that can handle data in the terabyte range. Tapestry offers the convenience, portability, and cost-effectiveness that traditional architectures simply cannot match. The popularity of the "as-a-Service" model in a variety of fields, including web applications and AI inferencing, is further evidence of the potential of this approach.

In the realm of scientific visualization, Tapestry represents only one of the simplest forms of parallel visualization. This dissertation aims to expand the capabilities of VaaS beyond their existing applications to now support data-parallel, multi-device, and hypothesis-driven explorations.

First, I investigate how to create data-parallel visualization services. While many scientific visualization codes are already data-parallel at their core, their presentation is typically as a task-parallel interface. Simply layering a cloud interface on top of the existing code is not enough to fully take advantage of data-level parallelism. To a user, there is no difference between two task-parallel interfaces, where one is data-parallel and the other is serial, besides the overall throughput of the service. However, by incorporating stateful information into the service, a VaaS could expose its data-parallel nature to the user or developer, improving the underlying visualization and making it more performant. I aim to explore ways of achieving this goal.

Second, I will show that VaaS can enable applications where each user can use multiple devices simultaneously. In the same way that using multiple monitors with a single computer can improve productivity, there is potential in using multiple devices for similar improvements. Expanding VaaS to multiple-devices per user needs additional capabilities for state management and service discovery. In this dissertation, I aim to study the use of multiple devices each with their own distinctive capabilities.

Third, I will show that VaaS enables abstractions for problems where the end result is not known, such as the hypothesis-driven exploration of a new dataset. Existing approaches to VaaS have had difficulty adapting to the changing needs of a user in this explorative context. Most existing VaaS applications use fixed processing and rendering pipelines. Expanding VaaS to support hypothesis-driven work requires dynamic and flexible computation. In this dissertation, I propose advancements to VaaS that enhance its capabilities for hypothesis-oriented tasks, enabling it to become a versatile solution for real-world problems in a variety of domains.

The remainder of this dissertation is organized in the following way. In Chapter 2, I cover the background and related work for visualization applications, services, and the new domains that I study. In Chapters 3-5, I describe improvements to VaaS that are necessary

to support data-parallel, multi-device, and hypothesis-oriented applications, respectively. In Chapter 6, I demonstrate a new application that integrates multiple scientific applications to create a VaaS that versatile and expressive. Finally, in Chapter 7, I summarize the potential areas of future development for VaaS and the new capabilities thereof.

Chapter 2

Background & Related Work

In this chapter, I provide a foundation for the development of the Visualization-as-a-Service (VaaS) architecture. In Sections 2.1 and 2.2, I summarize the history of and comparisons between monolithic and VaaS architectures. In Section 2.3, I describe characteristics of three domains that have been under-explored in VaaS. With these domains, I also present three representative applications that I will study in this dissertation. Each application is motivated by the challenges posed by rapidly growing and complex scientific datasets.

The last decade has seen an explosive growth in the size of scientific datasets, with terabyte-scale datasets becoming increasingly common. Despite their enormous potential for insights, such datasets have proven to be difficult to analyze and understand due to their sheer size and complexity. To address this challenge, visualization has become a critical tool for enabling researchers to explore and communicate insights with others. Complex scientific visualization has a need for powerful computing power.

However, the growing trend towards remote work has made it increasingly difficult for researchers to access the powerful computing resources needed for effective data visualization. With around 45% of all full-time employees working remotely in 2021 [95], many workers have had to rely on laptops or other portable devices with limited processing power and reduced access to high-speed internet. Therefore, the need for scalable and portable data visualization tools is more pressing than ever before.

It is necessary to not consider visualization in a vacuum and instead to consider the combination of visualization and interaction. A modern solution to accessible visualization

must be designed with interaction in mind. Both SciVis and InfoVis embrace interactivity as a fundamental aspect of the field, with interaction recognized as one of the two fundamental aspects in InfoVis, alongside representation. Therefore, it is clear that interaction must also be a core tenet of SciVis. The ability to adjust and manipulate a representation is crucial to the success of SciVis since the amount of data is too large to be understood without some form of aggregation. Without these techniques, static visualization images make the process of extracting insights and valuable information from data sets difficult. Overall, interaction techniques have been shown to play a crucial role in both InfoVis and SciVis in serving the needs of researchers and analysts who work with large and complex data sets.

In 2007 [110], a taxonomy of interactions for InfoVis was proposed that includes: selection, exploration, reconfiguration, encoding, abstraction, filtering, and connecting. This taxonomy applies equally well for scientific visualization in general, but even moreso to VaaS. Powerful VaaS must be designed with these interactions in mind to ensure versatility and expressiveness. In the following chapters, I will use this taxonomy to guide the design of each application.

2.1 Monolithic Application Architecture

It has been found that monolithic systems, once considered resource-efficient and scalable, have certain limitations when it comes to long-term scalability. Recent research has revealed that these drawbacks are summarized as a trade-off between deployment and scalability [18], which limits their ability to support more applications and workloads due to their design constraints. As hardware and software improve, there is a limited ability for monolithic systems to adopt these improvements without sacrificing stability of availability. The alternative solution is to adopt an hourglass architecture designed to overcome these limitations by separating the application into smaller, independent modules that can be scaled and improved independently.

One promising solution for remote workers is to move heavy applications from personal computers to hosted infrastructures, enabling them to access powerful computing from a device or laptop with limited processing power. Although remote desktop access can alleviate

some of the problems with using large datasets for visualization, it is still a single-user and single-device use case. This is best demonstrated by large applications like ParaView. In a remote desktop scenario, a single instance of the ParaView application is allocated for a single user. When that user is idle, the resources in use by that application are unable to be redirected for other users. A hosted service that can be used by multiple users can be more scalable and accessible. By using a hosted service, remote workers can collaborate on the same application simultaneously without being limited by their individual devices' processing power.

The shift towards web-based applications and the ubiquity of web browsers have resulted in the widespread adoption of client-server architecture to handle large datasets, as seen in applications such as ParaView [13], VisIt [30], and ViSUS [79]. This approach allows web browser clients to offload the burden of loading large datasets from the client machine to a remotely hosted server.

To address the issue of scalability, the adoption of a restrictive interface is crucial for improving the performance of monolithic systems. As such, it is often easier to scale architectures that are more dynamic and service-oriented, allowing different layers of the application to be scaled independently.

One significant factor influencing the computing landscape is the shift from owning and maintaining "on-premise" infrastructure to utilizing shared cloud infrastructure. The public cloud has made massive scalability accessible to developers in two critical ways. First, external management of cloud services eliminates the need for physical hardware and maintenance. Second, cloud services provide scalable flexibility by creating new instances during times of high traffic and shutting down instances during times of low traffic.

The move towards the public cloud poses several challenges for the migration of traditional scientific monoliths. First, server-side heterogeneity is often at odds with parallel frameworks for HPC. In addition to the heterogeneity inherent to shared infrastructure, the implementation of global synchronization in frameworks like MPI for public clouds is slower and less predictable [56]. MPI is written and commonly used on systems where all the computing resources are nearly identical. On a symmetrically configured platform, this assumption is valid as long as the machine in use is not shared. The use of virtualized

containers, such as Docker [4] and Singularity [63], provides a convenient way for replicating the kind of environment that HPC software expects. However, its use alone is not sufficient for migrating software to the cloud.

A platform's heterogeneity can be the result of multiple kinds of machines, of shared usage of a single or virtualized machine, or of a variable and changing number of machines. It is this last case of a changing, or commonly called "elastic," addition and removal of machines at runtime that is a primary challenge for scientific HPC applications that embrace the cloud.

2.2 Visualization-as-a-Service (VaaS)

The guiding principle behind VaaS is its ability to define efficient and standardized interfaces to essential services. In relation to monolithic services, the VaaS approach is to take a monolithic client-server separation to its logical extreme. In this context, VaaS is a growing trend in distributed computing systems that leverages the Hourglass Model's [18] scalability to support modern data visualization.

Leveraging its scalability, VaaS provides an effective solution to modern data analysis challenges in many domains. In particular, it allows for intrinsic task-parallelism, such as generating many small visualizations with different parameters simultaneously [91]. This task-parallelism can be used efficiently by clusters of servers. VaaS has been successfully demonstrated for intrinsically task-parallel applications, such as automatic data management and visualization in ManyEyes [109], visual analytics for cybersecurity in rDaas [46], scientific volume rendering in Tapestry [88, 86], and many other others. Cloud-based designs can improve scalability further by decoupling server-side components from client-side user logic. By making the server-side code stateless, VaaS can natively supporting multi-user scenarios, and allow for processing many concurrent user-requests in parallel.

VaaS interfaces can be deployed on various types of infrastructure, including self-hosted on-premise infrastructure and public cloud hosting providers like Amazon Web Services (AWS). However, other kinds of hosting are possible too, on any machine that is remotely or internally accessible by another device. In later chapters, I explore this flexibility by hosting a VaaS on a local intranet from a personal laptop. This universality of HTTP and the

web ensures that VaaS application development is accessible on all major operating systems, making it easy for anyone to create and use VaaS services, regardless of their device.

2.3 New Domains for VaaS

In this dissertation, I aim to explore the adaptation of VaaS to new problem domains, and evaluate the new functionality required for their implementation. To achieve this, I have selected representative problems in each domain and used them as case studies to better understand the problem space as a whole. Each chapter in this dissertation explores the design and implementation of a VaaS for each general problem domain.

Data-Parallel VaaS is a domain that has been under-explored. At its core, VaaS is highly effective for task-parallel problems because each task can be naturally expressed in the common request and response format. When each instance in the service is independent and capable of performing any task, a task-parallel interface has proven scalable and affordable [86]. However, a data-parallel interface does not have the same guarantees necessitating study. It is notable that the interface for a service can be task-parallel even if the underlying visualization is data-parallel which is commonly the case for physics simulations [84] or optimized rendering codes.

Multi-Device VaaS is an emerging domain that has yet to be fully explored. Traditionally, VaaS applications have been designed to enable independence between users and devices, even when sharing visualizations asynchronously [92]. In contrast, multi-device VaaS is designed to utilize the unique capabilities of each device to address a single visualization task, while still maintaining consistency across devices. Multi-device VaaS is most effective when pairing two distinct but connected data sources and visualizations.

Hypothesis-Driven VaaS is not a completely novel idea, as every visualization application aims to answer users' hypotheses and questions in some way. However, in this dissertation, I propose a VaaS that has no predetermined results or visualizations and is instead flexible enough to generate and answer novel hypotheses using visualization.

While some previous work has incorporated different visualizations, they are still limited to pre-existing visualizations [106, 51]. In contrast, hypothesis-driven VaaS allows for the exploration of new visualizations in response to users' evolving hypotheses. This process requires improvements to the VaaS architecture, specifically in areas of data management, modelling, and process orchestration.

2.3.1 Data-Parallel VaaS & Parallel Flow Visualization

Data-parallelism is often a technical detail within the software's architecture. This detail is rarely exposed directly to the user except indirectly when the software is faster or requires different resources. In this dissertation, I explore the concept of exposing data-parallelism to the user of a VaaS.

Existing web services that are data-parallel are often implemented in a simplistic way where every instance has a complete copy of the entire dataset. This can be described as the "data-duplicated" approach. One example that is used in software delivery is the use of alternative hosts that each have a complete copy of the files of the primary host. This simplistic method of parallelism helps to ensure fault tolerance in case one of the hosts goes down. It also helps improve the overall throughput because each host can is used independently and randomly. However, there is a need for a different kind of data-parallel service.

Many existing VaaS are based on the data-duplicated approach. A major drawback is in the memory costs of loading large datasets and the compute cost of processing that data. This is most evident when trying to use virtualized cloud instances which are memory- and compute-constrained by design. To leverage the public cloud for complex scientific VaaS, the service's design needs to be aware of where data is loaded and which instance is available. This can be described as the "data-distributed" approach to data-parallelism. In this dissertation, I aim to explore how to operate a VaaS where each instance has a different working set of data.

In data-distributed parallelism, there are unique runtime constraints on the clients and servers. Notably, it is necessary to support service discovery so that each client is aware of what data is available on each server. It is similarly necessary for each server instance to be

able to discover each other and query for the overall CPU load on those instances so that work can be distributed more evenly.

Parallel flow visualization is a challenging visualization task that requires a data-duplicated approach to data-parallelism. This task, which is also called particle tracing or streamline visualization, involves the use of large 3D volumes of 3D vectors. Importantly, the most interesting datasets for flow visualization are also the most turbulent, and the most difficult to predict access patterns. This dissertation specifically explores flow visualization of wind velocity vector fields over the entire planet.

For flow visualization, it is essential to account for the method of "seeding" the starting positions of all flows. Once seeded, the flow lines can be extracted by a spatial integration using the vector field. However, parallel flow tracing in large flow fields presents challenges requiring data-parallel and task-parallel approaches. Several solutions have been proposed to scale batch-mode parallel particle tracing, but the unpredictable access patterns of flow visualization makes achieving parallel speedup challenging. Moreover, interactive and configurable seeding methods are necessary to complement automatic batch-seeding methods for effective visualization of large flow fields.

In addition to automatic batch-seeding methods, interactive and configurable seeding methods are necessary to complement efficient visualization of large flow fields.

Algorithmic improvements such as data and task partitioning, dynamic load balancing, and runtime job and data management may optimize the process. It is a lengthy and difficult process to optimize flow visualization algorithms. Consequently, I do not attempt to optimize this optimization and instead focus my attention on the high-level design of instance-instance communication in the cloud which is the most broadly applicable to the general design of VaaS.

2.3.2 Multi-Device VaaS & Mixed Reality Visualization

Multi-device visualization is an approach that is gaining more support due to two major changes in the computing landscape. First, it is common for people to have 2 different devices on their person, such as a laptop and a smartphone, or a tablet and a smartphone. Second, it

is becoming more common for mobile devices to have a different and unique set of capabilities from other devices, such as a touch screen or native augmented reality (AR).

Use cases for visualization are usually constrained to a single device. Complex visualization is often implemented within a single application like VisIt or ParaView. Visualization on HPC systems is similarly limited to a single command line window or a single GUI application.

The availability of multiple devices each with their own capabilities has made a new use case possible: cooperative visualization across two different devices at once.

Mixed-reality visualization is a task that benefits from a multi-device approach. This is supported by the observations that augmented reality (AR) is better suited at visualizing and navigating 3-dimensional data than traditional screen-based methods [14, 20, 55]. Equally, AR is ineffective as a medium for screen-based interactions largely due to its lack of precision.

As AR technology becomes increasingly prevalent, it is likely to be an essential component of new interfaces. Since AR operates natively in a 3D environment that surrounds the user, it can more effectively convey 3D data than a 2D medium [14]. AR is additionally well-suited for presenting graphs, where it can display relationships and trends more effectively [20]. In addition to information presentation, AR holds promise for teaching new concepts [55], and has many other exciting use cases. Augmented reality also enables novel ways of interacting with data compared to surface displays. Multisensory interaction technologies may allow us to interact with data in more intuitive and thus easier-to-understand ways [77, 74, 71]. External devices like the MagicBook [23] and the Personal Interaction Panel [105] have also been explored for AR user interaction that requires higher precision.

Registration in AR has been a significant area of research and also a significant hurdle to its adoption. In the past, marker-based systems of registration meant that there was limited potential to move in 3D space without the device getting "lost." However, they posed usability and design challenges in creating AR scenes. With more powerful mobile devices, higher resolution cameras, and advanced sensors, software-based handheld AR has become more popular. Libraries such as Google's ARCore, Apple's ARKit, and Microsoft's HoloLens use sensor data from cameras, accelerometers, and even depth map cameras in

the case of HoloLens to enable real-time AR mapping which have enabled the use of AR in complex scenes without any prepared environments or objects.

The benefits of AR are also limited by characteristics of the medium. Guidance towards areas of interest in an AR application is important and challenging [43]. Further, some kinds of interfaces built for AR in inefficient or imprecise, such as voice recognition for specifying text or on-screen virtual buttons for selecting objects. These interfaces already have an efficient mechanism on ordinary computers: scrolling and tabbed interfaces for guidance, keyboards for text entry, and mouse pointers for selection.

In this dissertation, I explore the design for a multi-device VaaS that combines screen-based with AR interfaces. The result of this combination is an extension of AR called "mixed-reality" (MR). Control of the virtual world in AR often relies on interactions that aim to replicate real-world actions. Various studies indicate that AR is beneficial for tasks such as 3D object manipulation [62]. However, there are some interactions for which desktop applications are better suited, such as data filtering and querying [14]. Therefore, developers should be able to offer both forms of guidance in their applications [77, 74, 71].

Authoring content is a long-term research priority within the field of AR. In the past, some researchers have drawn inspiration from web content creation and developed markup languages for creating AR scenes [48, 9, 94]. Others have applied the design metaphor of hyperlinked web pages and established AR scenes as a web of interconnected areas [48]. Meanwhile, there are many proven distributed graphics applications that support multi-device and multi-user scenarios for massive multiplayer online games such as World of Warcraft, Minecraft, and Roblox and for collaborative analytics, as in Munin [15, 45]. Peer-to-peer architecture is the most robust and scalable strategy in this area.

As previously investigated, the integration of different types of devices, such as surface and mobile devices or AR and surface devices, can create coordinated spaces for individual users or collaborative teams. However, desktop or other surface screens remain more suitable for tasks, such as text entry and precision selection.

It is desirable to have a system for making it easier to integrate two different applications, runtimes, or datasets. Such a system should be able to support the common use cases of the platforms they originate. This means that it should support switching between multiple

browser tabs or coordinating the visuals of two different applications within the same virtual workspace. In Chapter 4, I study the implementation of a powerful and generic system for the management and synchronization of states between web applications on personal computers and AR on smartphones using a VaaS that is hosted on the user's own computer.

2.3.3 Hypothesis-Oriented VaaS & Large Graph Visualization

Hypothesis-oriented visualization is not a wholly new idea: every visualization is intended to answer some kind of hypothesis. Instead, in this dissertation, I consider the idea of a VaaS that has no predefined visualizations and is flexible enough to be used for testing hypotheses in both a qualitative visual and quantitative numerical manner.

For a visualization to be flexible enough to answer many kinds of hypotheses, there are unique constraints on its adaptation to a service. By design, the service must be capable of handling multiple datasets at the same time. There is also a need for a dynamic engine that can process user requests efficiently.

The encoding for a visualization request that is expressive enough to answer hypotheses is another challenge. In this dissertation, I explore the design of a service whose requests are complete programs in a domain specific language (DSL) that get executed on the user's behalf. This concept is supported by the understanding that a sufficiently complicated request for a visualization is itself a restrictive method of defining a program. The need for the service to be shared and limited in capabilities suggests that the programs must be finite either with the use of pure functions or the enforcement of runtime limits.

Large graph visualization is a task that benefits from the hypothesis-oriented approach. Graphs are a common means of depicting connections among entities. Commonly, graphs are visualized via graph processing algorithms first which summarize aspects of the graph into quantitative numbers and these numbers are used to understand the properties of the graph. However, there is another means of understanding graph qualities through visualization.

Several graph rendering packages exist, including Gephi [16], GraphViz [35], Cytoscape [100], graph-tool [80], and SciPy/Matplotlib. Moreover, graph databases such as Neo4j and fully-featured graph packages such as Spark's GraphX [42] and Stanford Network

Analysis Package (SNAP) [68] exist that offer both graph rendering and other capabilities. These tools are often limited in their ability to be iterative and dynamic. It is more common for the resulting visualization from one of these tools to simply be "the" visualization with very few controls to modify it.

The methods that applications provide to help users make discoveries about graph data are often manual or semi-manual processes based on the model of focus+context [102, 32, 25, 103]. A noteworthy example is H3Viewer [73] which can extract a minimum spanning tree from a 100K-node graph, and let users interactively control rendering of a multi-scale version of the extracted data. Similar to H3Viewer, many other graph applications rely on some technique to reduce the number of elements rendered at each point in time. For example, ASK-GraphView uses the notion of hierarchies and clustering to limit the number of rendered elements [7]. GraphMaps facilitates the browsing of large graphs in tile-sized portions [75]. Other works used querying [21, 37] and machine learning [28] to limit the size of the rendered data. In addition, there are also a plethora of existing works on how to integrate and handle streaming data [98], and/or to construct easily customizable analytics dashboards [97].

In this dissertation, I seek to address the challenges of large graph visualization without the need for filtering out the data ahead of time. The resulting system should be capable of rendering the entire graph at interactive rates.

The simultaneous need for performant quantitative measurements and qualitative visualizations suggests the use of OpenGL. OpenGL is the de facto standard for GPU accelerated rendering. With accelerated rendering, frame rate is no longer a bottleneck. Instead, the challenge lies in managing "visual cluttering," which is limited by users' cognitive abilities.

Although graph rendering is commonly assumed to be a problem constrained to 2-dimensions, the usage of OpenGL gives access to a 3rd dimension for new purposes. The many kinds of graph layout algorithms [53, 17] including Fruchterman-Reingold (FR) [38, 40], t-SNE [107], SFDP [52], ARF [41], LGL [8], and more. The design used in this dissertation does not assume or require any particular kind of layout. Instead, the goal is to be able

to flexibly alter the layout according to the user's needs. The implementation thereof is a system called Graph Shaders (GS).

Graph Shaders offer an additional level of control over OpenGL-based graph rendering for feature-rich graph visualization applications, enabling the creation of more powerful user insights. To satisfy users' expectations for high interactivity, graph shaders are designed to provide more expressive control over renderings and run natively on GPUs. However, it is important to note that GS graph shaders alone are not intended to be an independently useful system.

It is the combination of Graph Shaders and a stateful VaaS architecture that enables large graph visualization. The VaaS must incorporate statefulness to manage the complex software runtime environment involved in the user-defined and dynamic process of large graph visualization using graph shaders.

Chapter 3

Data Parallel Service

3.1 Overview

In this chapter, I study the design of a data-parallel VaaS for the problem of parallel flow visualization. Parallelism is an intrinsic property of VaaS. Although this is often in the form of task-parallelism, there is potential for a data-parallel VaaS to be effective. Exposing the inherent data-parallelism as part of the interface enables some unique opportunities for co-locating data repository services with data visualization services. In Chapter 6, I will demonstrate how the methods described in this chapter can be integrated into a larger framework. The work in this chapter was previously published as: Hobson, T., Hammer, J., Provins, P., and Huang, J. (2021). Interactive Visualization of Large Turbulent Flow as a Cloud Service. *IEEE Transactions on Cloud Computing (IEEE TCC)* [50].

Scientists organize their research around data. When a research community starts to continuously accumulate, curate and share community datasets, the community data repositories become a catalyst for future research. They are also a powerful source to engage the public, make science relevant, and deepen societal impact of science. Such continuously growing and open datasets are precious assets of the whole world. The Climate Forecast System (CFS) ensemble data repository [96] from NOAA National Centers for Environmental Prediction (NOAA NCEP) is a great example.

While cloud has already become the leading solution for creating distributed systems to support large data repositories, this chapter proposes to extend cloud-based functionalities

of a data repository beyond static data services. The primary proposal suggests interactive visualization services can become a component of cloud-based data repositories too.

Such a broadened scope of data repositories can help lower many barriers of adoption by diverse user communities. For example, researchers can accelerate their work by being able to always see the latest data to test and refine their hypotheses without needing to maintain an up-to-date local replica of the entire dataset. They can also collaborate with more people, by being able to share their findings with others who do not have, or cannot afford to have, an entire local copy of the data.

The ability to expand accessibility of powerful data visualization to thin user devices is a guiding principle within this dissertation. The VaaS in this chapter is centered around the idea of handling the complex and data-intensive computation of flow visualization on behalf of the user. This design enables devices with little computational resources to still participate in data visualization.

In this chapter, I focus on the design and implementation of a VaaS for interactive parallel flow visualization. Flow visualization is important to many disciplines, including atmosphere, ocean, fusion, petroleum, aerodynamics, and cardiovascular biophysics, where "seeing" the flow is often the first step in the scientific research process.

In addition, interactive parallel flow visualization offers a unique opportunity to study how to use heterogeneous cloud resources to achieve consistent parallel accelerations in support of synchronous user interactions. In contrast, traditional workloads of parallel flow visualization center around batch processing, which starts with a multitude of seeds placed inside a domain and then extracts the flow lines in parallel. While this mode of operation is a good fit within HPC environments, it lacks the flexibility and incremental cost efficiency of public cloud platform. It's impractical for community data repositories to each have and operate a dedicated HPC platform as well.

Due to this reason, I am focusing on the capabilities of public cloud platforms to make leading-edge scientific datasets interactively usable at an incremental cost. The prototype system is called Braid. Braid instances work collaboratively as a self-organizing swarm for parallel computing. Each swarm appears as a single cloud service, i.e. a Braid Service, which

can be used locally on an institutional cluster, or remotely on a public cloud such as Amazon AWS.

The results show that Braid Service has the following characteristics: (i) built for large flow data and ensembles of flow data; (ii) achieves fast interactivity; (iii) instantaneously available; (iv) serves multiple users concurrently; (v) serves users locally and remotely; (vi) supports a variety of user devices, and (vii) lightweight to integrate into applications.

Desktop applications can use Braid Service through a JavaScript library, *braid.js*, which transparently manages parallelism, performance, and fault-tolerance. By hosting NOAA NCEP CFS data in the cloud, this chapter shows (1) an application that can provide interactive visualization of 3D global atmospheric flow field to students (Figure 3.1); (2) a comparative visualization for scientists to analyze deviations between forecast vs. observed ground truth (Figure 3.8); and (3) a public-awareness application that integrates a Braid Service into the popular D3 [6] library. This last example enables any citizen to investigate how pollution from nuclear power plants in the United States, on any particular day in the year, can impact the entire globe (Figure 3.9).

All applications use a year-long observation dataset of the Earth's 3D atmospheric flow ($1,463$ time steps, $720 \times 361 \times 36$ spatial resolution, 150 GB) from the NCEP CFS repository [96]. The second application adds a corresponding forecast dataset of the same dimensionality from the same CFS repository. In all cases, the AWS setup required is less than \$0.70/hour.

The results suggest that cloud services, like the Braid Service, can conduct parallel computing using heterogeneous resources and support flexible, interactive, general use of large datasets on a desktop. These interactive use cases, coupled with efforts to quantify the exact cost-performance benefits of the cloud [36], expand the application potential of cloud hosted data resources.

The remainder of this chapter is organized in the following way. Section 3.2 describes the system architecture for the Braid Service. Section 3.3 demonstrates the application development process including how to integrate with Braid. Braid's scalability results are in Section 3.4.

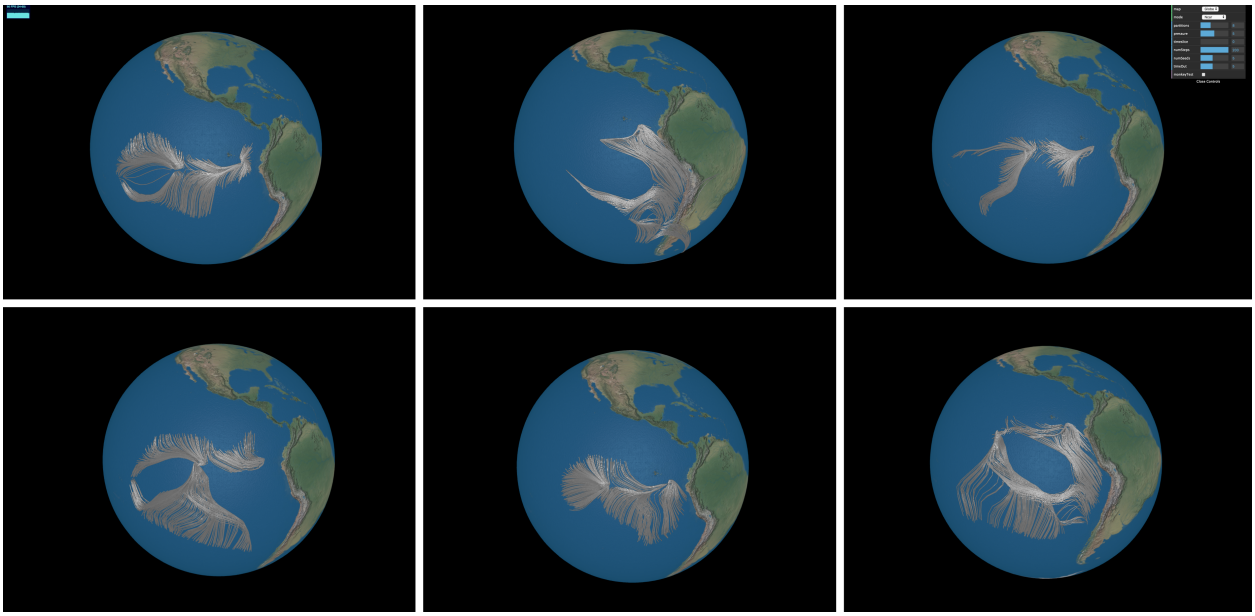

Figure 3.1: Example results using a web browser to interact with parallel flow visualization service supported by a self-organizing swarm of cloud instances. The system can support multiple concurrent users. Each subfigure shows $1,000$ particle traces of 200 time-steps each. They are extracted interactively from an observational dataset of global 3D atmospheric flow for the year of 2012; a total of 150 GB with 1463 time-steps at 0.5 degree geo-precision and 6-hour time-precision. The data is from the NOAA NCEP CFS repository. Exploration is instantaneously available in an on-demand manner. The service can use dedicated machines as well as public cloud platforms. When hosted on-demand on Amazon Web Services (AWS), for example, the total cost is personally affordable at less than $1/hour.

3.2 Braid: Design Considerations

When designing Braid, the complexities were separated into three categories: (i) those due to front-end interaction needs by domain scientists, (ii) those due to back-end parallel computing, and (iii) those required to bridge the front-end and the back-end. Accordingly, these categories relate to three distinct spaces: the application space, the system library space, and the swarm space. The three spaces are illustrated in Figure 3.2.

In this section, I discuss front-end and back-end separation, communication mechanisms, and related design decisions to make Braid efficient. At the core of Braid's design is its use of Docker Swarm. Accordingly, all references to "swarm" in this section refer to Docker Swarm which is characterized by its ability to dynamically increase or decrease the number of running services and transparently load balance between them. I discuss these more in Section 3.2.1.

Front-End. Application logic is in the application space, where the focus is user interaction and rendering. The system library space is concerned with accessing and managing interactions between the application space and larger-scale computing resources in the swarm space. Even though Braid uses JavaScript herein as the target language of the front-end, the separation of these spaces can be equally applicable to C/C++ or Python front-ends.

Back-End. The computing sources in Braid is a self-organizing swarm. Each instance inside the swarm is a Docker [4] instance. The notion of swarm is to highlight that there is minimum "cluster-level" orchestration, which makes the swarm model a more natural fit with cloud platforms like AWS. To this end, it should be noted that a swarm can just as easily run on a user's many-core workstations or small-scale clusters. The results account for both situations.

Communication. Braid has two distinct modes of communication. The first is for transient connections that need to be opened and closed on demand, which is primarily used within the swarm. These transient connections are implemented through HTTP. The second mode of communication is for persistent connections primarily between front-end and back-end, so that many requests can be made simultaneously. These persistent connections are

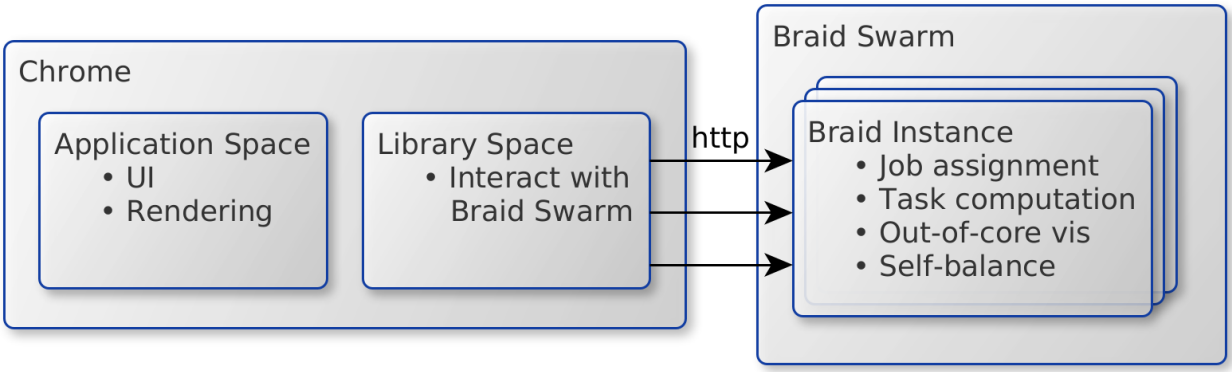

Figure 3.2: Braid System Diagram

through WebSocket. After a Braid Service receives requests through WebSocket, they are transparently transformed into HTTP requests that the swarm uses internally.

Lightweight System Design. Both client- and server-side designs are kept minimal to operate alongside other compute resources. In Chrome, the entire memory footprint, combining the application and library spaces, of an on-demand flow visualization application with functionality as in Figure 3.1 is less than 10 MB. The server-side swarm's data management uses a thrifty out-of-core scheme to lower memory footprint, typically using only 50 to 100 MB of memory in total. This greatly increases the overall system's portability.

Always Parallel Design. Even when there is a single front-end using a back-end swarm, the operation of the front-end and the back-end are both intrinsically parallel. To this end, instead of treating each user interaction as a request to be answered in a step-locked synchronous cycle, Braid treats user interactions as a continuous stream of asynchronous requests.

When a user moves the mouse on the globe (Figure 3.1), particle tracing requests are sent by Chrome to the back-end swarm as HTTP requests immediately and continuously. As Chrome continues to manage all outstanding requests transparently, the requests received by the swarm are distributed to Braid instances, which work together in parallel. During the process of extracting traces, incremental results are sent back to users. In wide area tests, the time to start receiving traces is under 0.1 seconds. *braid.js* running inside Chrome transparently manages the receiving of the parallel and continuous streams of extracted traces. Hence, although the application space makes a single-user assumption, the user always benefits from the multi-user assumption in the swarm.

Development Challenges. The parallelism inside of Braid offers benefits while it also presents several challenges. First, it is non-trivial to capture the global state of the swarm. For this reason, Braid uses distributed event logging in order to precisely record the state of individual instances as requests go through the system. Second, the dynamic nature of requests going through the system, and the related stochastic characteristics, makes it hard to exactly reproduce an exact flow of requests and events at runtime. Third, the same randomness in the system can introduce significant noise when benchmarking system performance.

3.2.1 Swarms of Braid Instances

Figure 3.3 shows an overview of the swarm architecture. The number of nodes and containers can vary as needed, even at runtime. Docker Swarm's manager receives and distributes incoming requests and is required on conventional computing clusters, but not required on public clouds like Amazon AWS, where the AWS load balancers serve the same purpose.

Each computing node runs a Docker daemon process, which interfaces with the Docker Swarm manager. There can be a variable and configurable number of Docker containers on each node. Each Docker container is a light-weight, fast to spin up, and self-sufficient virtual machine. Since the concept of "node" is virtualised on public clouds, the common hierarchy of Swarm-Node-Instance can be compressed to simply Swarm-Instance, although this difference is negligible for software development.

I refer to each instance as a Braid Instance and the entire Docker Swarm as a Braid Swarm. All Braid instances are identical. Each instance is a fully independent entity. HTTP is the only communication protocol used by Braid instances, regardless of communicating within or outside the swarm.

Every Braid instance runs its own HTTP server, responsible for receiving and queuing incoming requests. The swarm manager manages an overlay network between nodes. Braid instances on this network are able to freely communicate with one another. The swarm manager handles all DNS and routing within this network, which is not exposed externally.

The choice of HTTP as the sole communication protocol is to leverage the proven multi-threaded solution within modern web servers that can reliably receive, queue, and managed large amounts of concurrent requests. To my knowledge, few parallel visualization systems have similarly efficient, robust, and high throughput asynchronous communication capabilities.

The swarm manager receives and distributes incoming requests, but in no other way orchestrates parallel computation. The Braid instances self-organize to work in parallel. The only global synchronization is snapshotting swarm-wide workloads, so that instances can each adjust their own self-balancing scheme accordingly. The snapshotting operation takes place every 5 seconds.

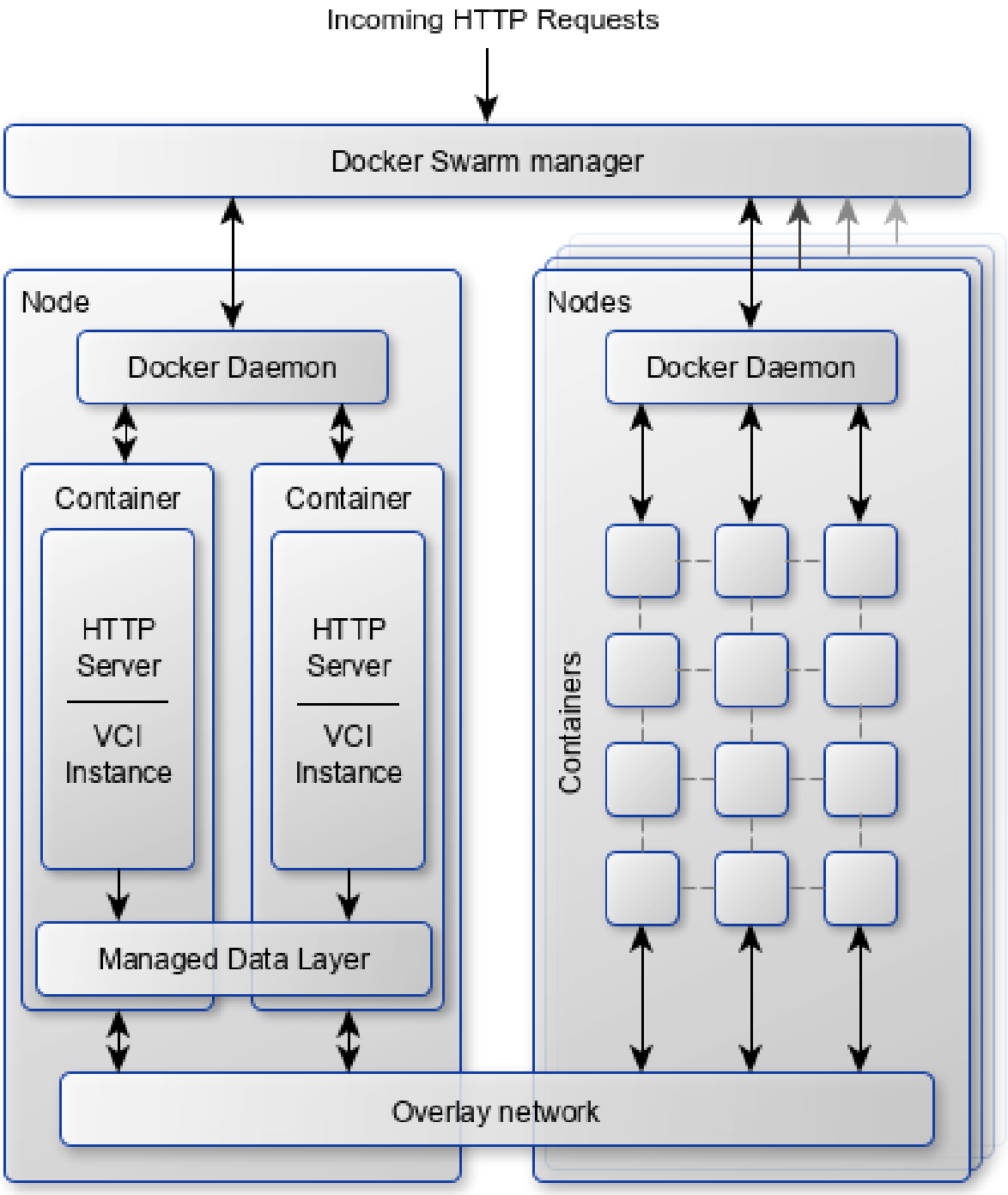

Figure 3.3: Braid Swarm Overview

The Braid Swarm has some resemblance to stream processing in how collaborating processing threads are used, because the Braid Swarm also routes requests through a network of operators to construct results. The main difference is that Braid swarm works on stored data, as opposed to streaming data; hence, key requirements for stream processing do not apply to Braid, i.e. query mechanisms and the need to keep data moving [104]. In addition, Braid Swarm is lightweight with a much smaller functional scope, assumes only standard Linux process management, without dependence on additional scheduling frameworks or resource managers. Lastly, in comparison to well-known stream processing systems [11, 111, 39, 76], Braid swarms run on resources that are minuscule.

3.2.2 Braid Instance

Inherently Threaded Design. The HTTP web server run by each Braid instance manages all computation and communication tasks of the instance. In essence, Braid uses a collection of web server processes for parallel computing. This design decision has two architectural reasons.

First, when parallel particle tracing is both data- and task-parallel, there are many disparate yet collaborative tasks to be managed [60, 83, 81, 72, 29, 59]. Although it helps to dedicate specialized threads for each specific function, managing a large number of threads scalably is non-trivial. The high-throughput thread management used in web servers can be reused as an efficient and reliable solution to this need.

Second, when using containers, a universal interface of collaboration is HTTP. In this case, there is a performance advantage, as well as a portability advantage, if parallel computing task can be mapped onto HTTP web server model directly.

Braid is built on an HTTP compliant web server in Python using a high-performance kernel in C. This design is easily adaptable for other HPC programming languages. I note two key details regarding threading mechanism here.

Threading vs. Forking. The internal design decision to use a threading server over a forking server is due to reasons of latency and shared memory address space. In particular, I/O is a fundamental challenge of all large-data visualization systems. In a data-parallel manner, each Braid instance is responsible for its assigned data partitions. The goal is that

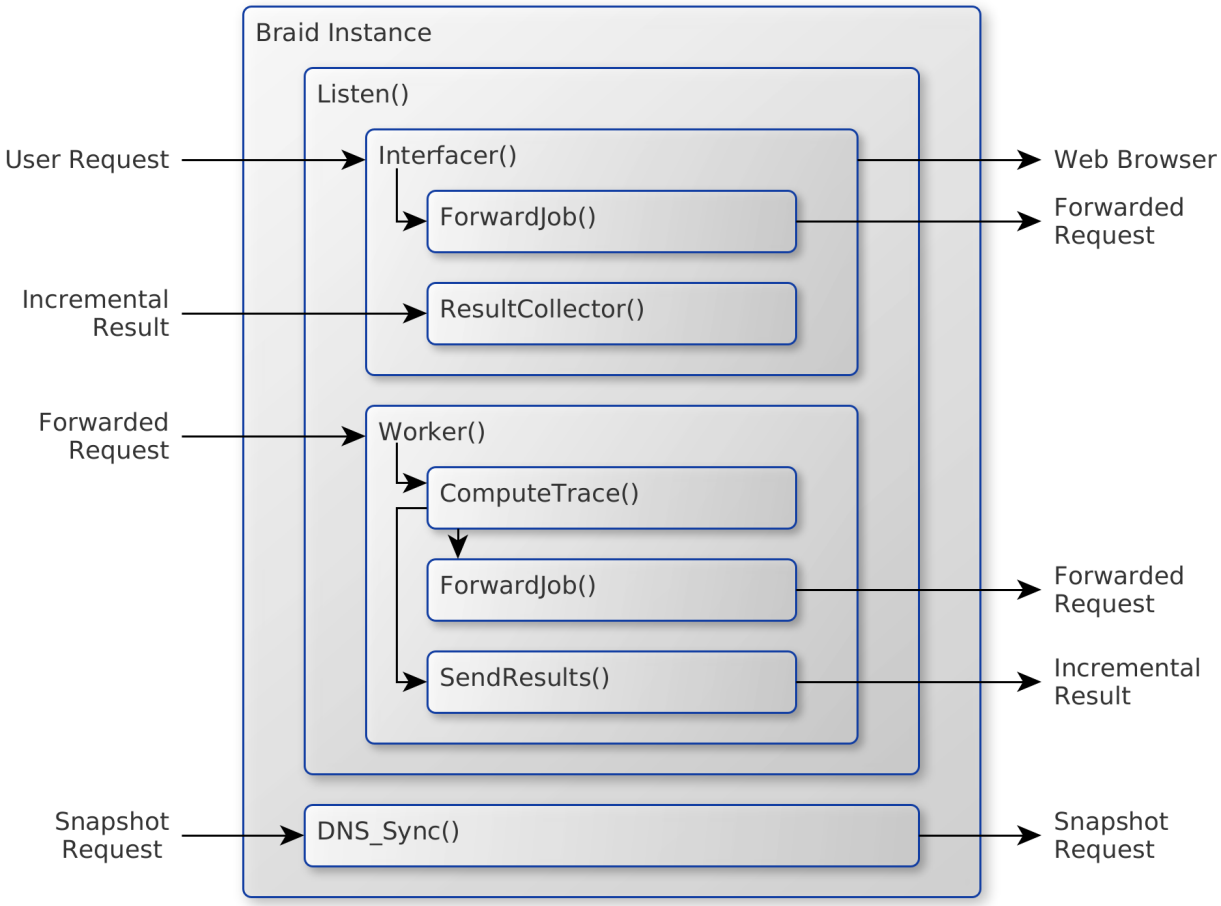

Figure 3.4: Braid Instance Components.

the data loading as well as resident-memory management parts of the instance can be shared by all threads. This way, a single out-of-core visualization implementation can minimize I/O operations as well as memory footprint for an entire instance.

My Braid instance is a derived class from the Python `ThreadingHTTPServer`, which spawns new threads for each request using `ThreadingMixIn` internally. Through evaluation many potential designs, this way of implementation offers the best efficiency, both for thread spawning and thread joins.

3.2.3 Specialized Threads

Braid's design philosophy comes from the domain of Unix: that is, to enforce rigorously that each thread is specialized in one type of task, and to ensure that threads can easily collaborate.

Figure 3.4 shows the internal parts of a Braid instance, where every box is a type of thread and arrows are HTTP requests being sent or received. These threads are based on `ThreadingMixIn` derived from Python's core `socketserver`. As threads have different functions, they have different lifespans too.

Listen(): this thread is up for the entire lifespan of the Braid instance. It runs continuously at all time and listens for new requests on the open socket. It does not perform any real task other than to determine the type of request and spawn off appropriate specialized threads to handle the requests. In particular, Listen() thread spawns off Interfacer() threads and Worker() threads.

DNS_Sync(): this thread is up for the entire lifespan of the Braid instance. It runs periodically to query the Docker swarm manager or the AWS load balancer to get an up-to-date list of instances as well as their load information. DNS_Sync() also provides the swarm membership information to any threads upon request.

Interfacer(): An Interfacer() thread is created by the Listen() thread when a client HTTP request is received. No Interfacer() threads have overlaps because they each serve a different client request. Requests from the same user are treated as independent requests and are handled by separate Interfacer() threads.

Interfacer() threads have the lifespan of a client request. When a user cancels a client request, for example, by closing their web browser, the corresponding Interfacer() thread is terminated.

Worker(): Worker() threads are created by the Listen() thread when an HTTP request is received from an Interfacer() thread. A Worker() spawns off a ComputeTrace() thread to perform the computation. As incremental computation results are ready, a SendResults() thread is spawned to send them to the ResultCollector(). Especially for flow line advection, the advection may not have reached the targeted length requirement if the flow line exits the assigned partition. When this happens, a ForwardJob() thread is created so computation can resume on the target instance.

3.2.4 Parallel Processing

As many-core processors become mainstream, processing power is getting condensed into smaller and smaller physical footprints. Not only are workstations with 48+ vCPUs very affordable, nowadays a cloud service with even 100s of processors can be affordably available on-demand at less than $10/hour.

Work Assignment and Data Partition. As with common practice, data is partitioned along the 4 spatio-temporal dimensions in a pre-processing step, followed by a 5-way partitioning with 4 voxel ghost regions resulting in 625 partitions. All partitions are on the same network mounted storage and are loaded by Braid instances at runtime as needed through memory mapping.

When a Braid instance is started, a set of partitions are assigned to that instance. These are *primary assignments*. As typical in fault-tolerant systems, replication (i.e. k-replica) is allowed, where k can be 1, 2, or 4. When k is larger than 1, additional partitions are assigned to each instance as *secondary assignments*.

Both primary and secondary assignments remain the same throughout the lifespan of the whole swarm. The mapping between partitions and instances uses a computable data assignment. This hash is determined by two parameters (the number of Braid instances and the number of partitions) and is based on a simple deterministic round robin process yielding a mapping between instances and partitions. Although there is not an explicit predetermined

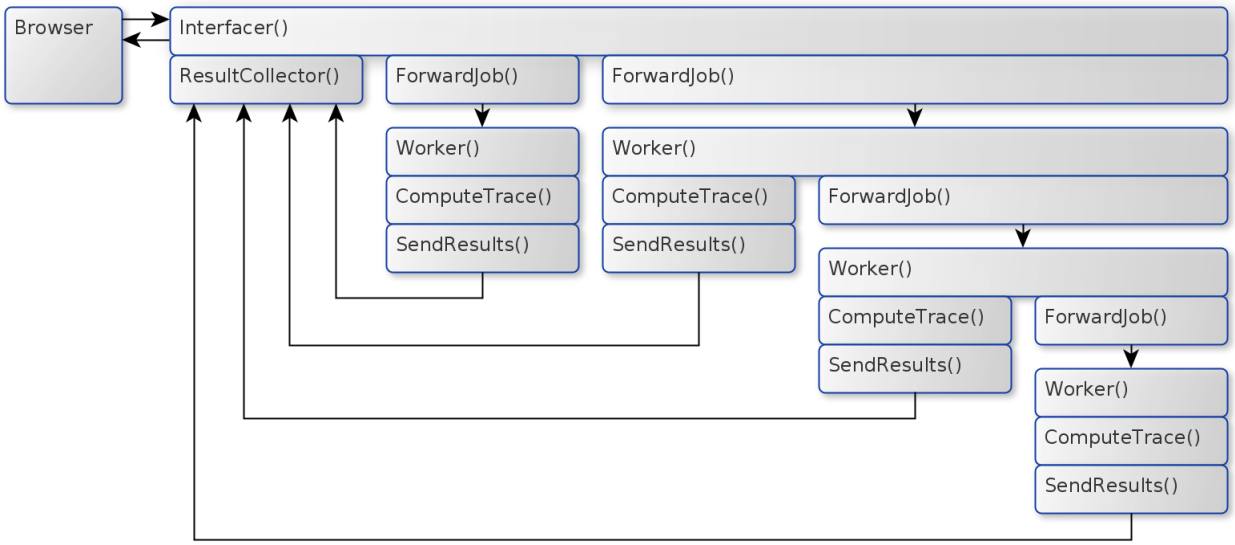

Figure 3.5: Workflow tree illustrating how a user request is handled. The verticality of each box represents serial execution while boxes horizontal of each other are executed in parallel.

partition assignment, each instance is able to compute the same list at runtime without any extra communication.

As particle tracing is computed, some particles may exit the assigned region for an instance. When this happens, that instance uses the data assignment mapping to identify candidate instances to continue the trace. These candidates are filtered based on load estimates (Section 3.2.6) and the particle is forwarded.

Workflow Tree. Through system-level routing (Section 3.2.6), a Braid instance is chosen at random as the main responder. The Listener() on that instance spawns off a dedicated Interfacer() thread for that request, which appears as the root of a workflow tree, shown in Figure 3.5. These threads may spawn other threads either within-instance (stacking) or cross-instance (arrows).

The Interfacer() thread will spawn off ForwardJob() threads, one for each of the target Braid instances, because they are assigned jobs according to swarm-level partitioning (Section 3.2.6). Each target Braid's Listener() receives the request and spawns off a dedicated Worker() thread to handle the computation.

A Worker() thread spawns a ComputeTrace() thread to handle the job. This thread always creates a SendResults() thread to send the newly computed incremental results. A ForwardJob() thread may also be created if the trace is not yet complete so the other instance can compute the rest of the trace.

Threads Spawning. The workflow tree depth depends on the expected length of flow lines, the number and distribution of the seeds, and how data partitions are assigned. To spawn many threads efficiently, I use Python's `ThreadPoolExecutor` as shown in Listing 3.1. The ForwardJob() thread uses a thread pool to distribute forwarded jobs to all of the Braid instances in parallel.

UUID. When Interfacer() threads generate jobs, it avoids collision between job identifiers using universally unique identifier (UUID), a proven concept in distributed systems and databases [64]. I use Python's `uuid.uuid4` function to generate UUIDs as 32 byte random strings, seeded with the process start time.

Working Set Minimization. Since cloud architectures incur much lower costs when instance sizes are small, it is beneficial to minimize the in-core memory overheads. Hence,

```
from requests import post
from concurrent.futures import ThreadPoolExecutor, wait

hosts = ['http://1.2.3.4:8840', 'http://1.2.3.5:8840']
seeds = [(0.0, 0.0, 50.0, 0.0), (80.0, 40.0, 50.0, 0.0)]
executor = ThreadPoolExecutor(max_workers=len(hosts))
futures = []
for seed, host in zip(seeds, hosts):
    kwargs = { 'url': host + '/trace/',
               'json': { 'seeds': [seed] } }
    future = executor.submit(post, kwargs=kwargs)
    futures.append(future)

done, _ = wait(futures)
for future in done:
    future.result()
```

Listing 3.1: Braid's server code for creating and managing runtime threads using Python's `ThreadPoolExecutor`.

```
void rk4(float *seed, float *newseed) { /* omitted */ }

int trace(size_t nseeds, float *seeds,
          size_t nsteps, float *trace) {
  #pragma omp parallel for
  for (size_t i=0; i<nseeds; ++i) {
    memcpy(OUTPUT(trace, i, 0), SEED(seeds, i), 16);
    for (size_t j=1; j<nsteps; ++j) {
      rk4(OUTPUT(trace, i, j-1), OUTPUT(trace, i, j));
    }
  }
}
```

Listing 3.2: Braid's particle tracing kernel with OpenMP parallelization.

during processing, each instance is designed to only load as small a spatial-temporal partition in the dataset as possible. To this end, in order to maintain generality, I decided to manage such data access patterns on the granularity of memory pages. This is done through memory mapping mechanisms provided by all modern Unix-flavored operating systems. As a result, rarely used pages get swapped back to disk while commonly accessed ones stay loaded.

Braid uses runtime communication, in the form of request forwarding across Braid instances, to ensure that each instance focuses on small partitions; whereas collectively the instances have a full coverage of the entire spatio-temporal domain with as small an overlap as possible. This is managed by having primary and secondary assignments for each partition, as shown in Figure 3.6. This reduces total in-core memory needs to 10's of GBs even when the full turbulent flow dataset amounts to 100's of GBs.

Scatter-Gather Design Pattern All communication in a Braid swarm follows the scatter-gather pattern and happens point-to-point without using explicit collective communication primitives like barriers. The request forwarding mechanisms scatter requests through the system as necessary and then the responses are gathered. Additionally, all instances poll their DNS_Sync() thread periodically for updates in the swarm-wide membership information, including whether there are newly added or removed instances.

3.2.5 Worker Thread Life Cycle and Results Streaming

Computational Kernel. The computation kernel of the Worker() threads is implemented in C to be usable from Python. The main computation task is flow line advection, using 4th-order adaptive size Runge-Kutta, modified from Numerical Recipes [82] to be thread-safe. I further accelerate the kernel using OpenMP as in Listing 3.2 to process multiple traces at once.

Interfacer() Heartbeat. A unique design requirement for Braid swarm is fault-tolerance in relation to the client's status. A client can cancel outstanding requests in several ways, such as through new interactions or through closing the web page.

The Interfacer() thread is the root of the workflow tree (Figure 3.5). Cancelled visualization requests trigger the Interfacer() thread and all of its within-instance threads to be terminated.

Across other instances, the corresponding Worker() threads need to be terminated too. This is done through the mechanism of a heartbeat signal. Specifically, while a Worker() thread operates, the Worker() thread checks the heartbeat of the Interfacer() thread.

The heartbeat information is collected by making an HTTP request to the ResultCollector() thread before running the low-level kernel. With this, the Worker() thread can detect that a user request is no longer valid and prunes the workflow tree accordingly.

Results Streaming and Termination. Two threads are involved in continuous results streaming: ResultCollector() collects partial results and Interfacer() sends partial results back to the requester. The former maintains a key-value store of all jobs created by the Interfacer() and tracks each job's status by their UUIDs. As ResultCollector() receives computed results from various Worker() threads, each job's status is updated, either marked as complete due to a termination condition being met, or marked as on-going. The termination conditions can include reaching the targeted length of flow line, or the flow line exiting the domain of the dataset.

The Interfacer() is the only thread that serves data back to the requester, and does so as soon as partial results are available. The application space, i.e. app.js, is the only place where the geometry of flow lines are assembled based on the UUIDs.

The ResultCollector() tracks per job completion status until computation for the whole request completes. The Interfacer() manages the start of the workflow and sends results with the UUIDs back to the requester. These two threads share a mechanism of double-buffering: one buffer for ResultCollector() in write-only mode, and the other for Interfacer() in read-only mode.

When request processing is completed, the ResultCollector() thread initiates the termination of the entire Interfacer() thread. I note that, in the normal scenario, no worker thread should be still alive serving that request. In other words, the entire workflow tree for that request should have only the Interfacer() thread, i.e. the root node of the tree, remaining alive.

3.2.6 Self-Organizing Swarm

Distributed Request Management. Request management is distributed across three tiers. First, the Docker Swarm manager, or the AWS load balancer, randomly assigns incoming user HTTP requests to a different Braid instance. Second, the Interfacer() thread created by the assigned Braid instance's Listen() thread polls the DNS_Sync() thread on the same Braid instance to get the most recent snapshot of system's operating status. The Interfacer() creates the jobs and decides how to optimize the mapping between jobs and all Braid instances. Third, individual instances make job distribution decisions in ForwardJob() threads. Each of these decisions seek to distribute system load as evenly as possible, as illustrated in Figure 3.6, each Braid instance can be assigned primary and secondary partitions.

Primary Partition. Upon initialization, each Braid instance is assigned a preprocessed partition as its primary partition. Altogether, the primary partitions collectively form a complete coverage of the entire spatial domain. There are no overlaps between primary partitions. The problem domain is partitioned by a grid pattern, each instance takes one cell from the grid.

Secondary Partition. In addition to the primary partition, instances may also be assigned other partitions to be used when at increased load. These partitions cover the same area as the primary ones, essentially forming another complete covering of the problem domain. When a job is forwarding to an instance based on its secondary partition, it is handled the same as it would be for its primary partitions. Secondary partitions are only used when the k-replicas (Section 3.2.4) is larger than 1.

Flexible Assignment. Data partitions are not pre-distributed to any Braid instance. Instead, the partitions are centrally stored and only mounted into an instance when the instance is created. The instances use OS-level memory mapping to access the data but do not pre-load anything until data is accessed. In this way, all Braid instances implement out-of-core visualization and use a very limited memory footprint.

Runtime System Snapshot. Periodically, for example every minute, DNS_Sync() queries the Docker Swarm manager or the AWS load balancer to get the latest information

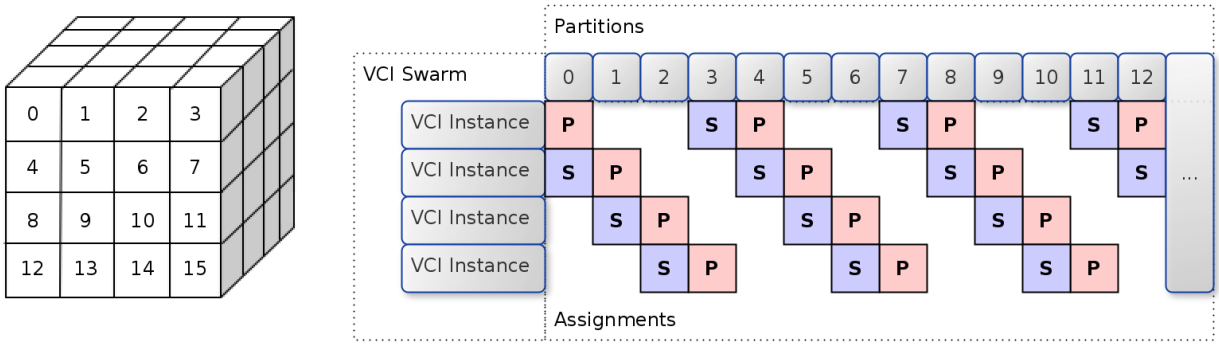

Figure 3.6: Round-robin partition assignment with $k = 2$ replication. (Red P) are primary partitions; (Blue S) are secondary partitions. Each VCI instance volunteers to cover its secondary partitions only when the corresponding primary instances are under load.

about swarm membership. After that, DNS_Sync() spawns off asynchronous threads to obtain and record the latest load information on all Braid instances.

When answering a status query from DNS_Sync(), each Braid instance reports the "ps" system load parameters, of which the CPU usage is the most important. This information is recorded in a per-instance hash, easily accessible by any function in the instance.

Thereby the DNS_Sync() is a specialized thread dedicated to keep track of how computing load is distributed among the entire swarm. Before any ForwardJob() threads make job distribution decisions, they query the latest snapshot from the DNS_Sync() thread so that their decisions adapt with variations in system load. Due to the polling rate of the DNS_Sync() thread, varying system load within 5 seconds do not factor into forwarding decisions which helps smooth momentary bursts of increased load.

Adaptive Load Balancing. For each job request, the ForwardJob() thread uses the same procedure to decide which Braid instance should receive the job. During the assignment phase, a round-robin mapping between partitions and instances determines which instance is responsible for that region, sorted so that the primary host is before the secondary hosts.

To decide the best host, the sorted list is traversed and the first one matching all criteria is selected. For load balancing, this criteria is a simple comparison between the CPU percentage and a pre-determined threshold. Braid uses 70% so that a very overloaded host does not get more load assigned to it.

Braid doesn't have a finite or pre-defined workload, hence pre partitioning is hard. As a result, it is about work distribution rather than work stealing, because the workload/request is unknown. Each unit of work completes within milliseconds as shown in the results.

The random data access pattern of particle tracing causes considerable performance challenges. The proven solution is to trade communication complexities to ensure good locality of computing tasks. Fortunately, Braid benefit from the freedom offered by virtualised containers. That is, accessing data via memory-mapping does not involve explicit traditional I/O operations, and at the same time includes a reliable resident memory management capability. In this way, minimizing memory footprint and providing effective caching is one of the same, which achieves great results.

3.3 Braid: Applications & Use-Cases

As popularized by d3.js [26], Vega-Lite [97], and other toolkit libraries, interactive information visualization inside web browsers is now a commodity. These toolkits have transformed the standard of portable interactivity for information visualization by managing the non-trivial interaction paradigm transparently.

For scientific visualization, portable interactivity is meaningless without data scalability. In this work, Braid's swarm space addresses data scalability, the library and application spaces address portable interactivity. Braid implements *braid.js* for the library space, and *app.js* as an example of application space, respectively. The design intention is that *braid.js* is a general reusable library, and that *app.js* will vary from one application to another. The separation between back- and front-end spaces enables independence and autonomy between them.

braid.js and app.js. *braid.js* is a compact front-end JavaScript library that helps developers to easily interact with multiple Braid swarms. Let's motivate the need to consider multiple swarms through a few use scenarios.

First, even for a single user using a single dataset, the incremental cost of $1/hr per Braid swarm makes it easy for a user to afford multiple swarms, which has both fault-tolerance and performance benefits. Such horizontal scaling is trivial in the cloud setting but very hard in traditional settings.

Second, data-enabled science is gradually gaining momentum. Maintainers and curators of a community-centric data repository may wish to stand up a web service for their repository. For example, the forecast component and observational components of the CFS data are obtained through different means and are updated at different intervals. If these are kept as separate Braid services, scientists can choose which services to use flexibly.

Third, when it comes to reproducibility, it is common to expect that a user may need to compare a new result with results from previous works. Practically, it has been very hard to reproduce another researcher's computing environment. If published results can all have separate web services, which may be spun up at very low costs, such comparisons can be prompt, convenient, reproducible and shareable.

For these reasons, *braid.js* always assumes there are multiple Braid services being used at the same time and transparently manages flow-control and fault-tolerance of each request in parallel. *braid.js* is highly concurrent but presents itself as a single-function interface. The basic settings include (i) the URL to each Braid swarm, and (ii) a callback function to execute when a response is received from each swarm.

Developers can customize their policy settings anytime. The key parameters are: (i) how many services are used at the same time; (ii) how long a single request can take before being counted as a failure; (iii) how many retries a request can have before stopping re-sends; and (iv) how long a single request with retries should wait before giving up and not re-sending. The default policy setting in *braid.js* is geared towards high interactivity (i.e. low timeout and low maximum attempts)

All services registered with *braid.js* are tracked individually. Specifically, *braid.js* tracks outgoing request traffic of each service, and receive, along with each result, the overall load of each Braid swarm. Depending on the policy choices set by the application developers, *braid.js* may add or reduce loads on each Braid swarm selectively. In each response, the CPU load of the service handling the request is returned. *braid.js* selects the lowest loaded service to dispatch the following requests to. This method guarantees natural load balancing for any task type performed service-side.

Upon sending each request, a configurable timeout can abort the request. If a request is aborted, *braid.js* tries re-sending the request until the max timeout or max number of attempts has been reached, according to the customized policy. When configured in multi-host mode, once any response is received, all other identical requests to other hosts are aborted. In addition, each service is monitored for failed or aborted requests. Continually failing services are dropped from the hosts list and no longer utilized.

When application developers make use of the Braid Service, application logic needs to be separated into an *app.js*. In the following sections, I show three examples, each targeting a different kind of user and a different use case. Accordingly, each application has its own *app.js*, which contains the front-end application that performs rendering and UI functions. For example, in a smooth tracking mode, *app.js* tracks mouse locations upon every single move, automatically maps seeding points onto the global map, and calls *braid.js* accordingly.

3.3.1 Application: 3D Flow Visualization

This application targets students who may be unfamiliar with flow visualization. By offering an interactive way to seed the streamlines, it enables students to experiment and learn. This application uses the JavaScript library THREE.js for creation, control, and rendering of a 3D scene graph using WebGL, and primarily maps data between the *braid.js* API and THREE.js.

Figure 3.7 shows the control panel of the application shown in Figure 3.1. The control panel includes temporal starting point of traces (i.e. "Seed Time"), the maximum length of flow lines (i.e. "nSteps/Trace"), the vertical height of seeding location (i.e. "Seed Pressure," measured in isolevels).

I also assume bundles of flow lines are more useful in turbulent flows. Hence, once a seed location is given, it is jittered by small random amounts (controlled by "nSeeds/Req") and sent together in a single Braid request. Obviously, a bundle can be reduced to a single flow line by setting "nSeeds/Req" to 1. In the tests, I set "nSeeds/Req" to 5. If *app.js* is handling 1,000 user requests/minute, 5,000 flow lines/minute are extracted.

Extracted flow line segments are received in the form of vertex arrays, which are then converted into stream tubes by using Catmull-Rom splines in order to allow for better illumination.

3.3.2 Application: Comparative Flow Visualization

Comparing models vs observations, or models vs models, is crucial for answering fundamental scientific questions. In climate research, such comparisons have led to a deeper understanding of climate extremes [34] and general atmosphere circulation [54]. Such comparisons are also indispensable in model validation [58] calibration [70], and scenario and sensitivity analyses [93].

As datasets become larger, such comparative studies are increasingly challenged by the data deluge, however. Interactive comparative visualization is becoming more difficult too.

The design of the swarm-service model was in part motivated by this need. Braid makes it easy to use specialized swarms to manage each large dataset separately, while the swarms

Figure 3.7: Left: UI controls of the tracing app (Section 3.3.1). Right: Monkey testing request locations. Each test makes $3,481$ requests in 1 minute at roughly a rate of 60 requests/second. Each request includes 5 seeds for particle tracing.

appear uniformly as standard web services. An analysis application can then have the feasibility and flexibility to adopt new datasets depending on needs, instead of depending on what a user's computer can manage.

To show this possibility, I compare how the NCEP forecast differs from the actual observations. To do so, *app.js* registers a second Braid web service with *braid.js*, and easily transitions to using two datasets at the same time. Each is 150GB in raw storage.

Figure 3.8 shows how the forecast model differs from actually observed wind velocity fields. The cool color shows forecast, specifically the `forecast3` data product of NCEP CFS. The warm color shows the observed data. The visualization shows that the model predictions have a more drastic shearing effect than the observed. The particle traces are seeded from January 1, 2012.

Braid's interactive visualizations in Figure 3.8 confirm global atmospheric circulation patterns overall. However, both the observed and the modeled patterns show unique variations on multiple levels. It is worthwhile for any scientific user to have such an interactive ability to investigate the scientific reasons behind model agreement as well as model divergence, which are general needs in all disciplines of computational science and engineering.

3.3.3 Application: D3 Flow Visualization

This application shows a possible integration with the popular JavaScript library, *D3.js* [26]. Its intended use case is for citizens without any training in flow visualization to be able to pose questions and answer them in an explorable way. This application used D3 because it is the de-facto standard for information visualization targeting general audiences.

This application prioritizes data analysis with multiple linked views. The flow seeding locations reflect 60 nuclear power plants in the US. The first view is a relational Sankey diagram, which describes the relation between individual power plants and the respective states the flow have travelled through. The second view is a line chart showing the number of states affected by a single power plant over time. The third view is a map that shows potential flow emitted from the power plants. A calendar selector is included with the third view so that users can flexibly experiment with daily varying atmospheric flow patterns.

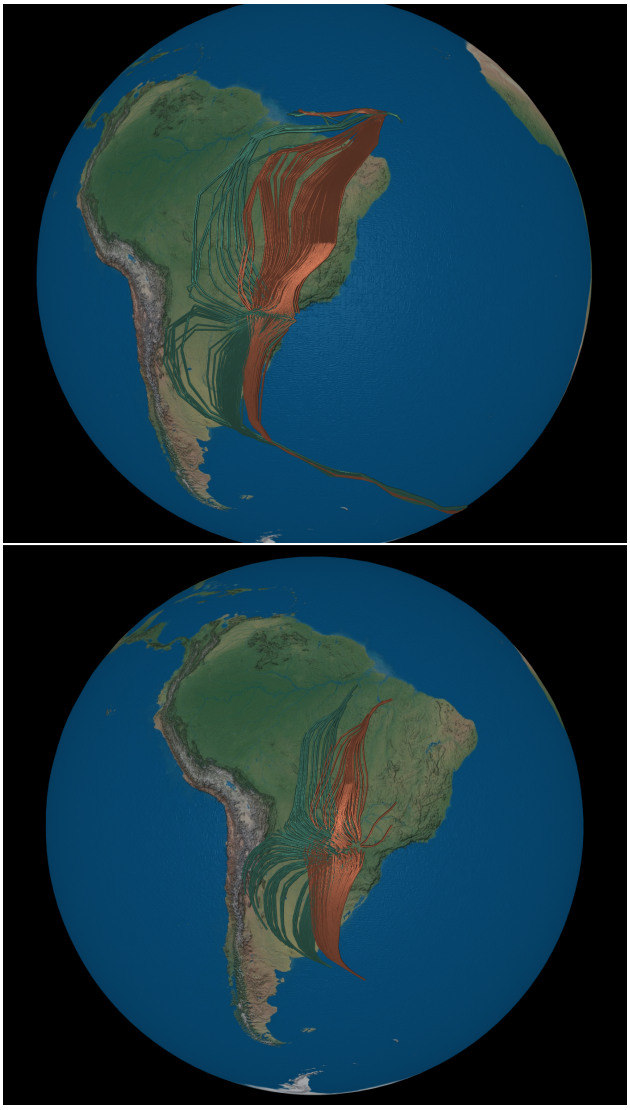

Figure 3.8: Comparative visualization showing differences between actual observation vs forecast atmospheric wind velocity patterns. Observation in cool color and forecast in warm color.

An example of the kind of insights enabled by this application is to examine the effect of a power plant disaster if it occurred on January 1st, 2012. In aggregate, it was found that 60 power plants have a combined coverage of effect on 37 different states. If one instead focuses on a single power plant in Michigan, then it alone is capable of affecting 9 different states in as little as 67 minutes. These analyses are very user directed and will depend on what exactly the user is interested in.

3.4 Braid: Scalability Study

Overview. Test Dataset. I chose the NCEP CFS atmospheric community dataset [96] because it represents the leading edge of assimilating observational data together with model predictions. My results are collected using a subset of this dataset that spans the entire year of 2012 at a 6-hour time interval (i.e. 1,463 time measurements), a global spatial resolution of $720 \times 361 \times 36$, along the dimensions of longitude, latitude, and pressure isolevels, respectively. This dataset has 3 attributes: latitudinal wind velocity, longitudinal wind velocity, and vertical wind velocity.

For most results in this work, I use a 150 GB piece of the CFS data which is based on the actually observed wind velocity. Results in Section 3.3.2 show interactive a comparative visualization between both the observed and the forecasted data. The additional forecasted component amounts to 150 GB of raw storage as well and has the same format as the observed data.

As a pre-computing step, data is partitioned, where each partition is $144 \times 73 \times 8$, for a total of 625 partitions. A ghost zone of 4 voxels has been added along each dimension to each partition because I use adaptive size 4th-order Runge-Kutta to compute the flow advection. Each partition amounts to 679 MB, whereas the partitioned CFS forecast dataset amounts to 417 GBs. The same applies to the observed data as well.

Interactive Test. Since this paper is concerned with portability, I test wide-area reliability via pre-recorded "monkey" tests, which have been recorded to emulate typical user interactions. This style of testing is effective at evaluating complex applications [108]. The test generates $3,481$ requests in a hand drawn path which traverses the globe over a

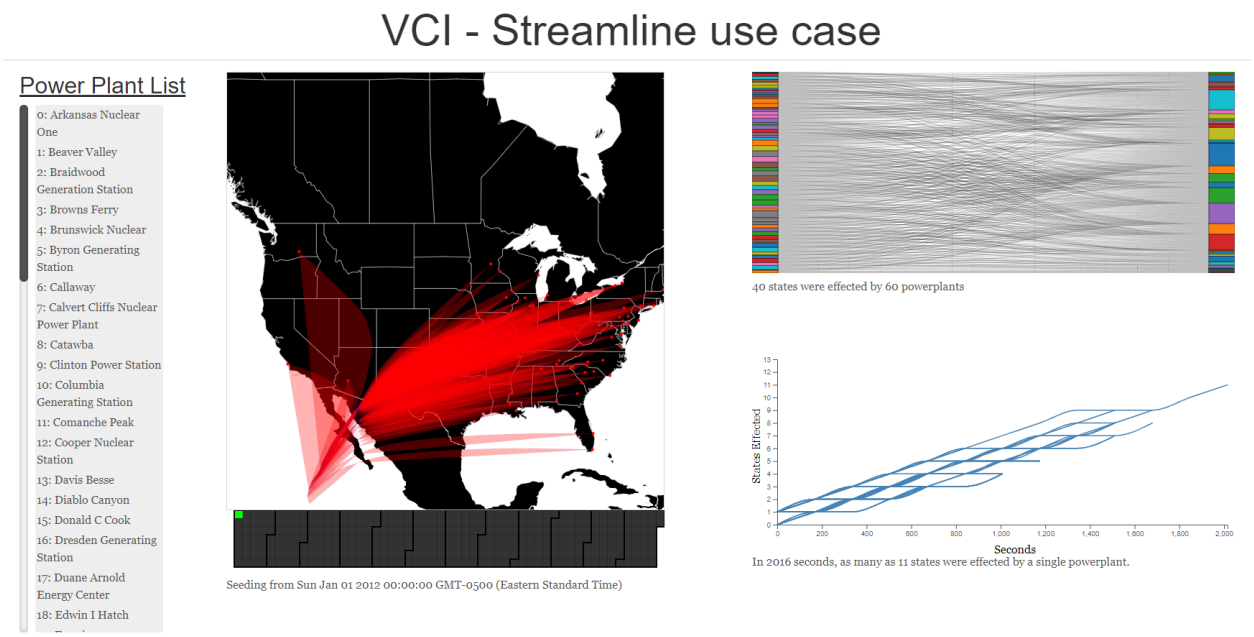

Figure 3.9: D3 application showing flow emitted from US nuclear power plants on January 1st, 2012. Left: Power plant selection list. Middle: Map view. Middle Bottom: Calendar date selection widget. Top Right: Sankey relational view. Bottom Right: Line chart view.

period of 60 seconds. This test is run once for each configuration. Average request rate is about 60 requests/second. Each request includes 5 seeds for particle tracing. Data is collected on the back end utilizing the logs generated by Braid which catalog request across the distributed system.

The test path is shown in Figure 3.7-Right. On average, each flow line has a length of $1,548$ miles and covers a timespan of 67 minutes. Each also uses data from several partitions of the dataset: 59.9% use only one partition, 45.5% use two partitions, and 1.6% use between three and five partitions. This shows that the monkey testing path exercises the system adequately as nearly half of all requests need to be load-balanced within the Braid Swarm.

Stress Test. I also designed a stress test, where each simulated user sends R requests at a time and maintains R active requests at a time by sending a new request as soon as the previous result has been received. The stress test scales up the number of simulated users as well as the value of R in order to understand the scalability of the Braid Service.

Test Logging. The results are measured after the tests have run. To do this, I collect detailed traces of the execution of the swarm including specific messages, variables, and wall clock times. Each trace is from the context of a single user request from the browser and this context persists across forwarded jobs. By measuring the time between particular messages, it is possible to determine how much time is spent in the tracing kernel or making cross-instance network requests, all without having to rerun the test cases.

Metrics. I collect performance metrics from the perspective of the client: latency until receiving the first byte of the result (Time To First Byte **TTFB**) and the last byte (Time To Last Byte **TTLB**). The ping latency between client and server is also collected, as well as the success or cancellation status for each request. Further discussion is in section 3.4.

Interactive Test on Dedicated Server. I first evaluate how swarm configurations affect various trade-offs among factors that include latency, throughput, system footprint, replication factors (i.e. k), I/O, and potential failures. All evaluation tests are run with the client being connected to an adjacent server. The testing client has an internet connection capable of sustainable rates of 500-550 Mbps down and 425-475 Mbps up.

For this purpose, I chose to use a rack-mounted machine to deploy a single swarm. The machine has dual Intel Xeon (E5-2650 v4, 12-core, 2.2 GHz) processor with a total of 48 vCPUs and 128 GB memory. I test $n = 4, 8, 16$ instances in the swarm and $k = 1, 2, 4$ replication in the swarm. These configurations are denoted as n-k, specifically, 4-1, 4-2, 4-4, 8-1, 8-2, 8-4, 16-1, 16-2, and 16-4. The per test case results are shown in Table 3.1.

I seek an optimal trade-off between request latencies and request throughput. Request latencies are measured by: time to first byte vs time to last byte, i.e. **TTFB** vs. **TTLB** in Table 3.1. The Braid swarm's total bandwidth to handle incoming requests throughput is measured by peak megabytes-in, i.e. **MB/s-In Max** in Table 3.1. The data shows a few patterns.

First, TTFB and TTLB are consistently correlated, with the best case TTFB and TTLB being 0.064s and 0.099s. Between the client (on residential Internet) and the centralized rack-mount server, the round trip ping time stats are: min/avg/max/stddev = 0.0590s/0.0658s/0.0871s. Hence, if a user sets up a Braid swarm for on-premise use, I estimate the TTFB and TTLB improve by 0.059s or more, resulting in a TTFB around 0.005s which is capable of sustaining 120 frames/second of interactivity.

Second, Braid instances require sufficient processing power to function optimally, because each Braid instance is massively threaded. The data shows that allocating 12 cores to each Braid instance is unnecessary. Braid instances with 6 vCPU cores or even 3 vCPU cores can function very well and deliver low latencies.

Third, increasing replication factor k does not yield increasingly higher performance, although it does show a minor improvement. As is common in many fault tolerant applications in the wide area, $k = 2$ replication is often used.

In Table 3.1, there are two memory use metrics. *P.I. Base Mem* is the per-instance base memory, measured as total memory use after the container is started but before performing any tasks. That value is consistently around 17 MB. *P.I. Net Mem* is the average additional memory used by Braid instances during the 3-minute long monkey test at roughly 60 requests/second.

Across the board, it was found that the memory-mapped lazy management of data does offer an efficient implicit solution to out-of-core visualization. *P.I. Net Mem* stays under 50

Table 3.1: Braid's Dedicated Single-Node Performance (WebSocket)

Test Case	P.I. Core Cnt	P.I. Base Mem	P.I. Net Mem	MB/s-In Max	Avg kIOPS Read	Avg TTFB	Avg TTLB
4-1	12	17.3	42.0	0.40	3.80	0.180s	0.252s
4-2	12	17.3	39.5	0.40	4.50	0.178s	0.246s
4-4	12	17.3	47.0	0.37	4.29	0.174s	0.251s
8-1	6	17.3	25.5	0.53	4.43	0.082s	0.138s
8-2	6	17.3	25.7	0.54	5.04	0.082s	0.139s
8-4	6	17.3	32.7	0.51	4.97	0.082s	0.138s
16-1	3	17.3	16.9	0.57	5.38	0.064s	0.099s
16-2	3	17.3	19.6	0.61	5.80	0.072s	0.121s
16-4	3	17.3	24.0	0.62	5.62	0.070s	0.113s

Table 3.2: Braid's Dedicated Single-Node Communication Comparison

Test Case	Method	Cancel	Complete	Med TTFB	Med TTLB	Avg TTFB	Avg TTLB
4-2	HTTP	1288	1951	0.201s	0.232s	0.357s	0.384s
4-2	WS	340	2448	0.056s	0.092s	0.178s	0.246s
8-2	HTTP	1086	2152	0.262s	0.287s	0.394s	0.415s
8-2	WS	204	3139	0.048s	0.078s	0.082s	0.139s
16-2	HTTP	1155	2071	0.323s	0.350s	0.424s	0.445s
16-2	WS	237	3192	**0.046s**	**0.074s**	**0.072s**	**0.121s**

MB throughout the tests. This is significant because such a footprint allows easy portability on virtually any system, even when using a Braid swarm as a background process on a user's personal desktop.

P.I. Net Mem drops as n increases because each instance is responsible for fewer partitions. When $n = 16$, *P.I. Net Mem* reduces further to less than 20 MB. Increasing k does increase *P.I. Net Mem* incrementally but non-linearly.

At $k = 4$ in Table 3.1 is the highest *P.I. Net Mem*, with a corresponding increase in I/O overheads, *kIOPS-R*. The latter measures thousands of I/O operations per second which signifies higher amounts of data loading. The relatively worse *TTFB* result for higher k may be due to the increased I/O and worse caching performance within each instance.

Table 3.2 compares Braid's two modes of communication: HTTP and WebSocket. I chose to test only $k = 2$ for these comparisons, as it is a commonly used redundancy factor. It is immediately apparent that WebSocket provides a higher success rate as well as faster response times in all cases by a large factor. This speaks volumes to the benefits of a persistent connection to the server in continuous-demand applications. The $n = 16$ WebSocket test case is of particular interest, as it manages not only the fewest cancellations of all tests, but also the lowest TTFB/TTLB. This suggests the WebSocket implementation scales very well with increased instance count.

It's worth noting that monkey test, as done here, is a typical stress testing technique in web-scale systems. The use of a total of 3,481 user requests, each with 5 seeds, is aimed to identify worst case scenario of Braid swarm. A typical user exploration does not come close to this level of system stress.

Interactive Test on AWS Server. As a significant test of deployability, I tested 5 different AWS configurations across two hardware classes: general-purpose, "t3," and compute-optimized, "c5." All instances are cloud-managed, where the base hardware is probably shared by many users.

I repeat the same monkey testing as with the dedicated server and based on Section 3.4, I test only $n = 8, 16$ instances and $k = 2$ redundancy. The network between the

Table 3.3: Braid's AWS Variable-Node Performance (WebSocket)

Node Type	# Nodes	CPU Cnt	# Instances	Total $/hr	Failure %	Avg TTFB	Avg TTLB
t3.xl	2	8	8	**$0.33**	41.45	0.252s	0.295s
t3.xl	2	8	16	**$0.33**	32.55	0.268s	0.343s
t3.2xl	2	16	8	$0.67	29.53	0.203s	0.249s
t3.2xl	2	16	16	$0.67	2.27	0.171s	0.275s
c5.4xl	1	16	8	$0.68	**0.09**	0.074s	0.096s
c5.4xl	1	16	16	$0.68	**0.09**	0.072s	0.095s
c5.12xl	1	48	8	$2.04	**0.09**	**0.068s**	**0.085s**
c5.12xl	1	48	16	$2.04	**0.09**	**0.066s**	**0.083s**
c5.24xl	1	96	8	$4.08	**0.09**	0.080s	0.104s
c5.24xl	1	96	16	$4.08	**0.09**	0.082s	0.107s

client and the us-east-2 AWS instances has a round trip ping timing of: min/avg/max = 0.0495s/0.0518s/0.0600s. Table 3.3 shows the testing results.

Since there are much lower performance guarantees on AWS than on a dedicated server, I define request failures as any re-sends or request with TTFB over 1 second. I report failure rates as **Failure %** in Table 3.3. The failure results clearly show that the "t3" class is not usable for my purpose.

Using "c5" instances, some patterns from Section 3.4 are again confirmed. Specifically, as long as a swarm has enough instances to ensure incoming bandwidth, further increasing instance count does not lead to lower request latency. Anticipating user's workload is key when configuring a Braid service.

Based on testing use, an 8-instance swarm on c5.4xl for $0.68/hr is the best choice. The more expensive instances performing worse is a repeatable result. A potential reason is that larger instances on AWS are shared by more users who have more sustained heavy loads. Again, the purpose of testing on AWS in this work is to show portability. The best performance for future users will be to use Braid swarms using their own on-prem cloud.

Figure 3.10 breaks down average server-side latencies into: (i) computation time, i.e. time spent inside computing kernels, implemented in C with OpenMP acceleration; and (ii) communication, i.e. the time spent doing network I/O for better job placement. The t3 configurations perform worse due to its general-purpose CPUs compared to the c5 configurations.

Stress Test on Dedicated Server. I ran stress tests that varied two parameters: (i) the number of users U, and (ii) the number of concurrent requests R. Accordingly, I design Test A, where U is varied while $R = 6$ ("many user"), and Test B, where R is varied while $U = 1$ ("many request"). In each of the stress tests, every request is for 1 flow line.

The results of Test A are in Figure 3.11. This result highlights that Braid is scalable to many users all abiding by a request limit like web browsers have. In this way, if every Braid user limits their number of active requests, then all users can make interactive requests at a rate of 80 requests per second.

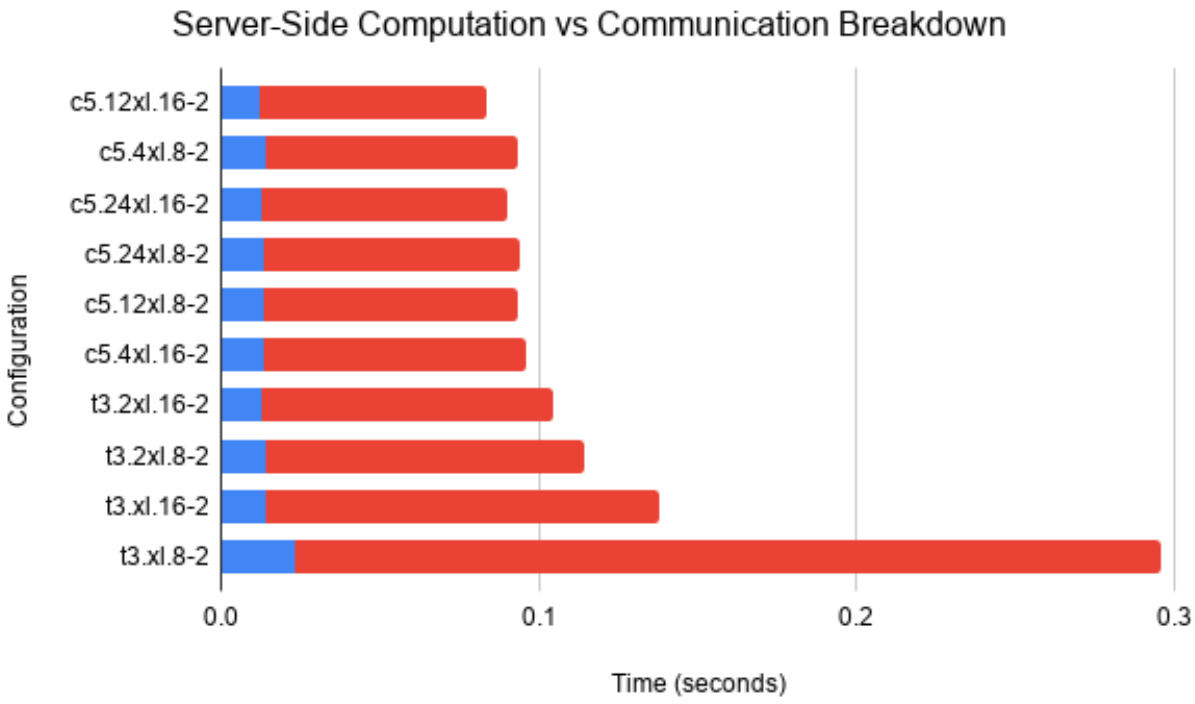

Figure 3.10: Evaluating how much time an average request spends doing computation (blue) vs communication (red) of each AWS configuration. The per-request tracking is enabled by Braid's design and use of UUID. The data shown is averaged over all requests in the monkey testing process. Reducing communication cost for better job placement can be as important as reducing computation cost for better efficiency. Experimenting with different configurations can help discover such trade-offs.

The results of Test B are in Figure 3.12, showing that higher degrees of performance are achievable by making more concurrent requests than a web browser can make. This illustrates the utility of WebSockets from a user perspective, as users are able to make up to 125 requests per second.

Discussion. Evaluating Braid's performance is tricky because Braid uses a custom-made web server for parallel computing. While works in parallel visualization evaluate efficiency and scaling from an algorithm perspective, the most applicable performance metrics for Braid are those from the literature of web systems, such as in [78] and [33]. Here, response time and incoming system bandwidth are the two primary metrics. These are reported as MB/s-In and TTFB (Table 3.1 and Table 3.3).

Client-Side Interactivity. Using a Docker swarm on either AWS or a dedicated server, the recorded average TTFB time is below 0.070s. In addition, in either server setup, the server can sustain answering 60 requests/second. On the client side, this translates to 60 frames/second interactive rates with 70ms latency. With typical graphics applications considering interactive use as in the range of 10 to 20 frames/second, I feel Braid is sufficient for practical use in single-user cases.

Computational Efficiency. An important paper [24] in IEEE Cluster'2020 benchmarked different parallel flow advection algorithms on Cori, one of key new HPC systems at NERSC. Due to how the user requests are generated, the only algorithm possible, which is also benchmarked in [24], is Parallelize Over Data (POD). The authors reported that seeding flow advection from a small spatial box is the worst case scenario for parallel flow visualization, due to great difficulties with runtime load balancing. For that use case, the authors reported POD to achieve between 70k-700k Steps Per Rank Per Second (SPRPS), when tracing streamlines that are 1000 steps each. The dataset used is NEK5000 flow of about 400GB. The computation is batch only, and far from meeting the latency requirements of interactive use.

In the stress test, the Braid Service plateaus at around 125 interactive requests per second for one user, using a 24-core dedicated server. Each request in the stress test is for a flow line of 50 steps. This corresponds to 125×50 steps per second, i.e. roughly 0.26k steps per

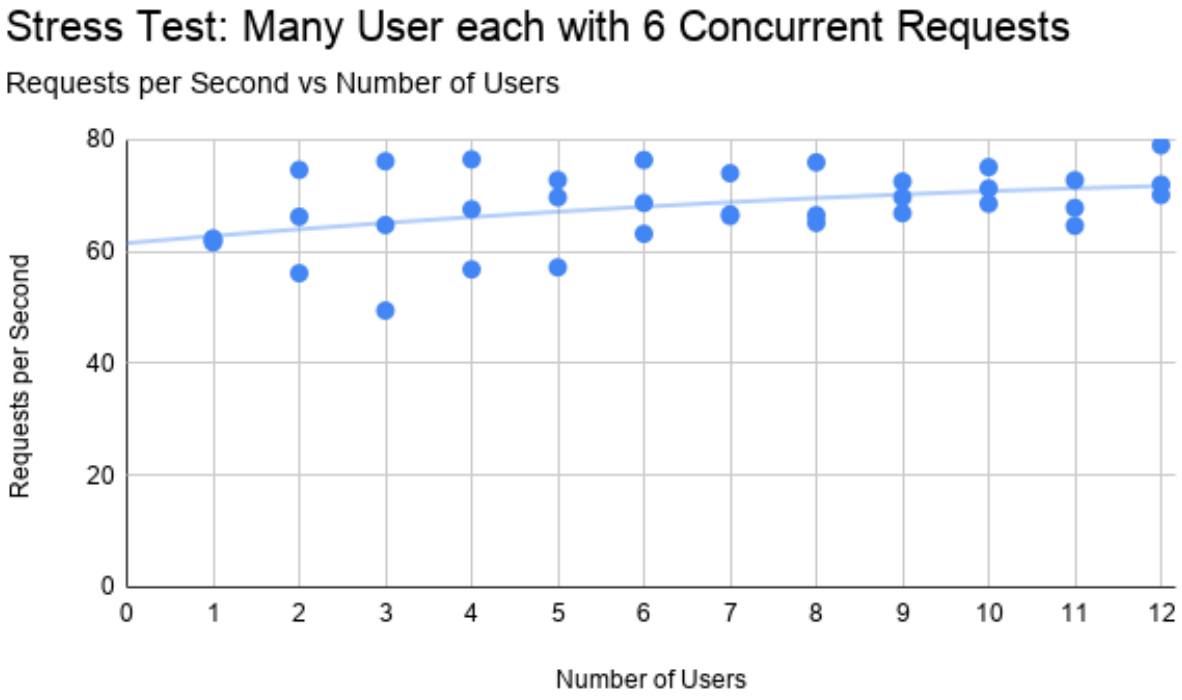

Figure 3.11: Stress Test A that shows how the Braid Service handles increasing number of users where each user only makes 6 concurrent requests at once, like in a web browser.

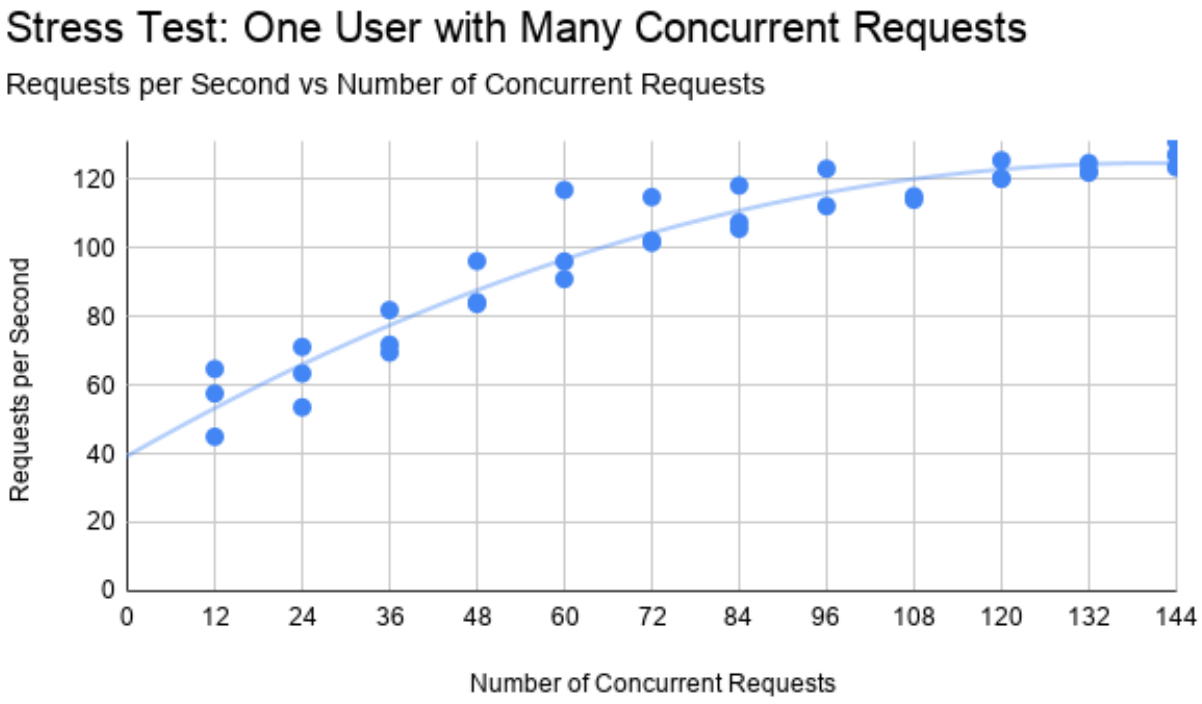

Figure 3.12: Stress Test B that shows how the Braid Service handles increasing number of concurrent requests from the same user.

core per second. I also designed a test with 10k requests, 5 flow lines per request, 100 steps per flow line. Ran the test 3 times through the Braid Service on the same 24-core dedicated sever. The average completion time was 23.01 seconds. This amounts to 9.05k steps per core per second.

Since Braid swarm is Python based and uses HTTP communication in order to run on the cloud, I feel this level of performance is acceptable, because Braid services can be deployed in an on demand manner for interactive use of a 150GB data set at fractional costs.

Chapter 4

Multi-Device Service

4.1 Overview

In this chapter, I present the design and implementation of a multi-device VaaS capable of upgrading web applications with an augmented reality (AR) companion interface. This functionality is transparent to the underlying application, requiring no source code changes. The work in this chapter was previously published as: Hobson, T., Duncan, J., Raji, M. Lu, A., Huang, J. (2020). Alpaca: AR Graphics Extensions for Web Applications. *2020 IEEE Conference on Virtual Reality and 3D User Interfaces (IEEE VR)* [49].

In Chapter 6, I expand on the work in this chapter by showing how the distributed data model can be used generally to manage stateful information across VaaS requests. The specific work on AR described in this chapter remains limited by the overall capabilities of phone AR. I expect that the technology behind widely accessible AR will improve over the next years, but at present, it is still too limiting for general use. In future chapters, I will instead demonstrate how this work serves as the basis for flexible state and data management within distributed VaaS services.

Since its inception, the web has led to a rich and diverse ecosystem for developing applications that impact the daily lives of every person. This ecosystem lives within 2D screens. With the advent of AR, researchers have attempted marrying AR with web in custom-built applications to leverage the rich interactions and explorability of AR.

Rewriting web applications with AR in mind can be costly. There are few overlaps between the web development and the AR development worlds. The near ubiquity of web browsers and the availability of a standard executing environment contrasts sharply with the niche and disparate ecosystems that AR applications must conform to. In essence, adding AR to an existing application has until now meant a costly rewrite and rearchitecture.

In this dissertation, I have developed a bridging framework that simplifies extending web applications into AR without a complete rearchitecture. The framework is called Alpaca and has two main areas of improvement. In terms of **content**, it enables immersive exploration of natively 3D information using an AR device, rather than being constrained to the 2D-screen-space. In terms of **architecture**, Alpaca provides a minimal bridging interface that reduces the complexity and overhead of mixed-space applications.

While some web applications may have AR counterparts, mixed-space web applications developed with Alpaca run on a desktop and an AR device (if present) simultaneously, which extends the spaces in which a user can explore the content in the web application.

Alpaca is intended to be a personalized bridging framework that connects live web and AR contexts on a user's laptop and AR device, respectively. At present, the visual channel of AR is the sole focus of this chapter. The design of the Alpaca Server which implements cross-device synchronization, is to run on the user's laptop rather than run on a publicly accessible service. By running the server locally, the cross-device latencies can be minimized with the intention to improve the user experience of interacting with a mixed-space application.

For AR developers, the Alpaca framework provides an abstracted, easy-to-use interface for developers who are not experts of DOM (Document Object Model) to effectively instrument and manage changes to DOM elements. The resulting multi-device capability aids AR developers in significantly lowering their barrier of entry into the widely varied ecosystem of web applications.

For web developers, the Alpaca framework provides a simplified interface and runtime, where code using the web graphics API (e.g. THREE.js) can be automatically bridged to run on an AR device and remain synchronized with the web application. Without needing to program on the AR device, user interactions with the scene graph can auto-update the corresponding web DOM elements in real-time.

For end users, Alpaca appears as a browser extension. When invoked, the Alpaca Server runs as a daemon process and transparently manages interactions between the DOM in the browser and the scene graph in the AR runtime. Users get a synchronized multi-device experience and can interact with Alpaca-enhanced applications both on their laptops and their AR devices, such as iPhone X.

The design of Alpaca have been driven by three types of application needs: *scene-heavy* applications have large complex scene graph models; *asset-heavy* applications require Alpaca Server to manage many assets concurrently at runtime; *update-heavy* applications incur a sustained high volume of synchronizations.

The design of Alpaca has been inspired by works such as MapReduce [31] and Tapestry [89, 87] that provide a minimal decoupled interface between the framework's components. The results in this chapter show that it is possible to use a common infrastructure to transparently abstract away all the cross-device communication and synchronization functionalities that are required in event-driven programming. Consequently, the AR device becomes an extension of the web application and becomes application agnostic, enabling reuse.

The framework is composed of three parts that work in tandem: a browser extension library, a state management server, and an AR device runtime. Each part is in charge of mapping from one type of data model to another. The browser extension maps the DOM to a JSON representation of the 3D scene to be used on the AR device. The server manages this JSON scene structure and synchronizes this scene to the AR device. The AR runtime then takes this scene and manages the rendering and interaction with these objects. AR-initiated events are handled by taking the opposite route.

The efficacy of Alpaca as a reusable infrastructure is demonstrated by building three types of applications: asset-heavy, update-heavy, and scene-heavy applications. These correspond to the Google Books, YouTube, and a GeoMap based application. The results include performance, memory footprint metrics of Alpaca, and a demo video (in supplemental materials) that show how a user can use the interoperable capabilities to effectively mix the 2D desktop space together with the 3D physical space made available through the AR devices. In a way, the resulting mixed-space is a new extension of the typical workspace of a

user, which deserves much future research by the computer graphics and the VR communities in general.

The remainder of this chapter is organized in the following way. Section 4.2 describes the design of each Alpaca component in detail. Section 4.3 discusses the design and implementation of 3 separate Alpaca applications. The results in Section 4.4 are evidence of Alpaca's generality and scalability for each of the 3 applications.

4.2 Alpaca: System Design

Alpaca is designed to provide a personalized experience for the user through multiple, cooperative devices on a shared local area network. Let's first illustrate the use case (Figure 4.1).

To a user, Alpaca's bootstrapping process is as follows: (i) on standard web pages, the user interacts with the page with no change; (ii) on AR-enabled web pages, a notification appears informing them that they can use an AR device for added functionality; (iii) the user opens an app on their phone that starts Alpaca AR runtime and sees AR contents enhanced from the web page.

The user then has two views of the application: on-desktop and in-AR. The desktop views generally remain as they were originally to remain familiar with the user. However, the AR view can be totally novel and supports experimentation into the realm of cross-device use. Some uses include: manipulating the data model of an application for new views of the data (YouTube example); mapping natively 3D data to actual 3D (GeoMaps example); and expanding the user's workspace to incorporate more data (Google Books example).

The extension library exposes listeners to the DOM so that the extension can be alerted when the application state has changed. This enables the development of reactive applications that automatically update based on the changes the user makes in the desktop application without modifying existing application's JavaScript code. The other two main APIs exposed by the extension library are: updating the AR SceneGraph rendered and creating AR listeners on SceneGraph objects. Together, these form the multi-device utilities needed for AR extensions. They will be described more in Section 4.2.

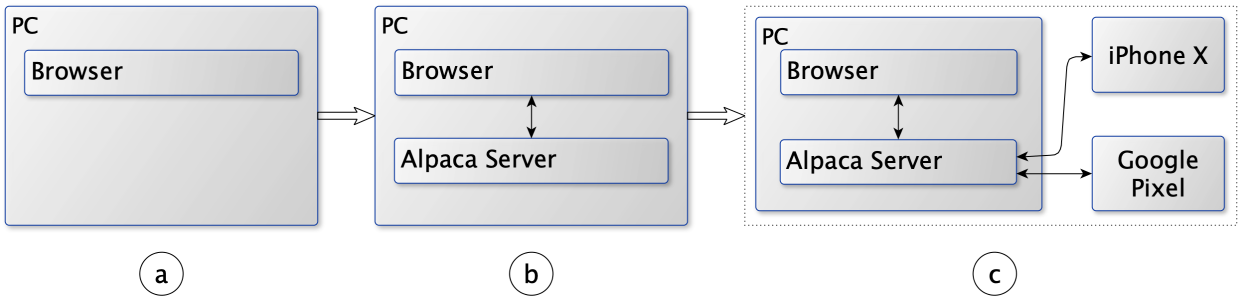

Figure 4.1: The basic workflow an individual user follows when using Alpaca. In (a), the user interacts with the web application like normal. In (b), the user enables Alpaca which starts up the Alpaca Server and the DOM/Event listener setup process (see Figure 4.3). In (c), the user starts the Alpaca AR Runtime to see and interact with the AR scene.

The Alpaca Server runs in the background and awaits HTTP requests from both the browser and the AR runtime. The server maintains an object store of stateful information and makes that store accessible. In addition, it provides a websocket interface that allows the server to notify clients when an object has been modified.

The AR runtime creates a websocket connection to the Alpaca Server and waits for updates to the SceneGraph object in the store. This object is the one created by the extension code. The runtime repeatedly renders this SceneGraph and waits for user interaction to trigger events that will be sent back across the object store to the extension, triggering the callback functions it registered.

To extend a web application to AR, a developer only needs to write the JavaScript extension using THREE.js to place objects in the AR scene. This code is injected into the web application when the user invokes the extension, after which, Alpaca transparently manages cross-device communication and state management.

For performance testing, the focus is on the following metrics: (i) AR startup latency, (ii) AR frames per second (FPS), (iii) Alpaca Server footprint (i.e. server memory usage), and (iv) application development complexity (i.e. lines of THREE.js and Alpaca API code). These metrics are discussed in more details in Section 4.4.

Overall Architecture. Figure 4.2 shows a more detailed view of Alpaca's system architecture. Alpaca is designed using industry-wide standard technologies, including Python's `aiohttp` library for the Alpaca Server, JavaScript's THREE.js library for the SceneGraph implementation, and built in HTML5 libraries. To the developer, Alpaca is accessible like any other HTML5 library for ease of integration.

Alpaca has three main components: Alpaca Server, In-Browser Runtime, and On-Device Runtime.

The Alpaca In-Browser Runtime is triggered through a Chrome extension in the browser. It is application dependent. The Alpaca Server runs persistently and is both application- and user-oblivious. In fact, all demonstration applications in this work use the very same Alpaca Server instance. The server can run on a remote machine for many users and applications or on the user's machine itself for a better guarantee of performance. The On-Device Runtime

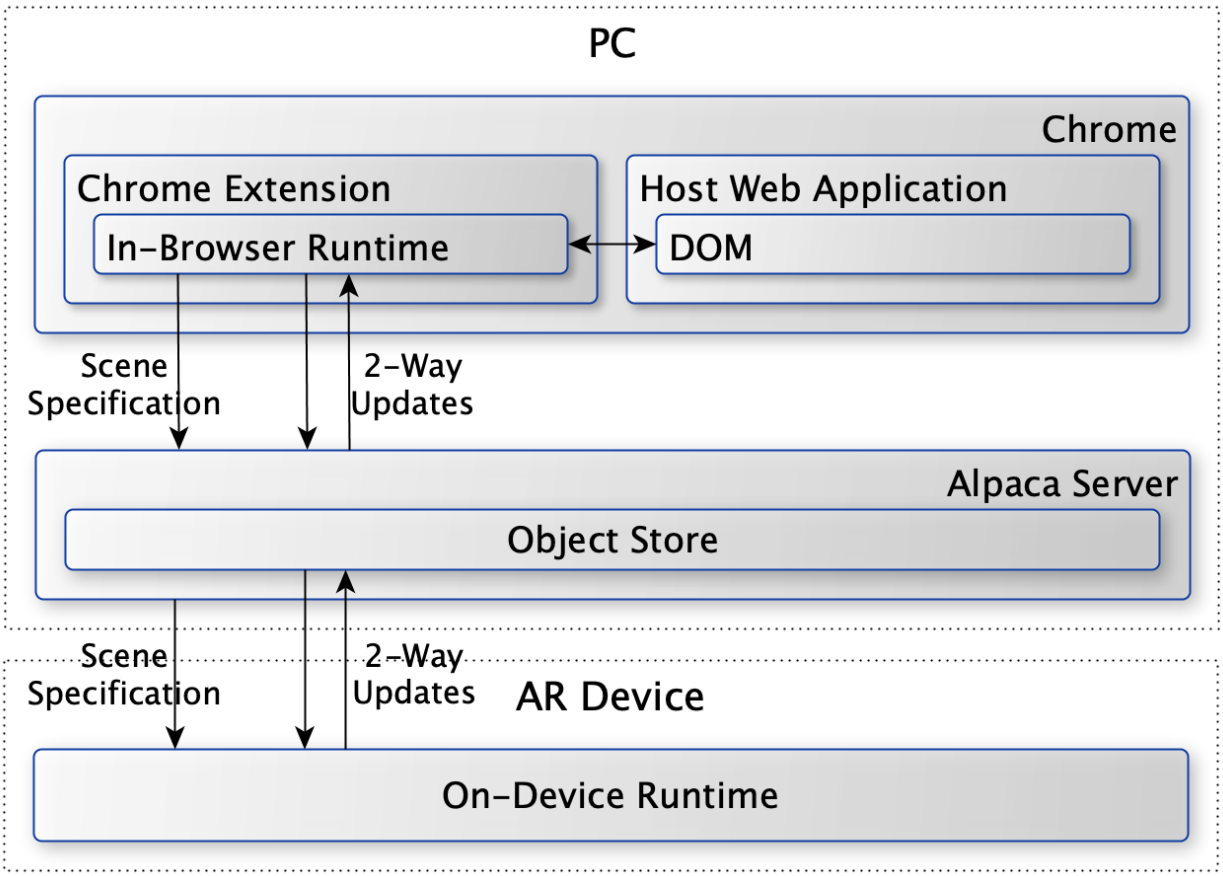

Figure 4.2: System architecture overview. The Alpaca Server provides the communication gateway between the web application and the On-Device Runtime and can reside on the same machine as the Chrome browser or a remote machine.

is application-oblivious. When a user's AR device is on, the device runtime queries the server for the scene specification, and performs the rendering and user-interaction tasks accordingly.

Host Application. The DOM is a key underpinning of all web applications [dom]. In essence, the DOM is just an XML model that describes all elements that a web browser renders and presents to each user. While creating an AR extension for a web application, absolutely no changes are made to the host web application.

Alpaca In-Browser Runtime. Alpaca's on-desktop presence is wrapped within a browser extension. This browser extension has different JavaScript scripts that can be injected into the web applications and provide the application-specific changes.

On first load, the Chrome extension contacts the Alpaca Server to register which and how a selected set of DOM objects in the corresponding web application should appear and behave in the on-device AR world. These Alpaca objects are web-session specific. For example, two users using the Google Books extension separately may have the same kinds of objects registered, but those objects are completely unrelated as they exist within different instances of the Alpaca Server. The extension also includes calls to the Alpaca In-Browser Runtime which manages event listeners, triggered when an action is performed in AR.

Alpaca Server. The server handles all stateful information that must exist between the in-browser and on-device runtimes. It also serves as a shared information space for any content that needs to be stored for longer periods of time. The Alpaca Server is application oblivious. Its stateful information management is also agnostic of content types, regardless of persisting images, canvasses, text, or other MIME types.

One Alpaca Server can support many applications and users concurrently. For example, all applications in this work actually use the very same Alpaca Server.

Alpaca On-Device Runtime. The on-device runtime is a single application that runs on the AR device to support MR content from the Alpaca Server. This runtime performs registration and automatically refreshes the AR scene when it has been updated by the in-browser runtime. User actions that should trigger state changes on the host application are sent through the Alpaca Server.

Alpaca's on-device runtime is application oblivious. Upon application start, the on-device runtime queries the Alpaca Server for AR objects and begins to automatically set up the

scene according to the specification registered by the corresponding Chrome extension. After first start, the on-device runtime will continue to operate, even if the in-browser environment is closed, because all contents sent to the AR device come from Alpaca Server.

AR Driven Design Considerations. AR helps us expand the sense of space from 2D to 3D. The physical world then becomes the user's native operational environment. Using Alpaca, the goal is to build a bridge between the desktop-side web world and the AR enabled human world. As a starting point, the current goal is on extending a user's current workspace.

Applications on AR face many general and known challenges of AR [85]. The main tasks on AR are: (i) accurate registration, (ii) efficient rendering, and (iii) reliable event handling. Through extensive experiments, four dimensions were found that influence the design choices behind Alpaca.

Performance. This limitation primarily comes from a need for a high frame rate to decrease user sickness. Without dedicating significant amount of resources, it is also hard to ensure uniform quality and performance across all devices. To improve performance, applications need to limit the number of polygons, and use image-based rendering wherein complex geometries are replaced by low-poly objects with textures.

Alignment. Humans can detect misalignments between virtual and physical objects easily. Better hardware can improve accuracy, just like how HoloLens uses depth cameras and the iPhone X and Google Pixel all move the alignment process closer to hardware. Application scenarios matter too. The discovery of usable surfaces is left to the OS-level software of the AR device; for instance, ARCore and ARKit use image-space techniques to detect surfaces.

Interaction. AR is not better than desktop on all interaction tasks. Some interactions are really very well suited for the desktop medium, for example, drop-down menus and precise selection. While some researchers have been designing new hardware to bring fine-tuned controls into the physical world to improve the precision of control in an AR application [22], researchers have also been improving software detection to make more usable controls [66]. In this regard, Alpaca provides a new software approach by allowing developers to easily

integrate desktop and AR interactions in the same application. Alpaca integrates with the device OS for native AR interaction.

Mobility and Cost. Many AR systems are not portable due to size, weight, and system installation needs. Some high-end AR systems are not widely available due to cost as well. Alpaca is still a web-first infrastructure. It is also designed to be device agnostic.

In-Browser Runtime. The Alpaca In-Browser Runtime (IBR) needs to be able to run on any supported websites. It is made of two parts: the user-defined code and an Alpaca-provided library of support code.

In order to avoid requiring a developer to have to modify an existing web application, and also allow the in-browser runtime to be packaged and easily released, the in-browser runtime is implemented as a Chrome extension. As a web-standard practice, each extension, when triggered, checks whether this current webpage matches a whitelist of applications that Alpaca manages. If there is a match, then the in-browser runtime is injected as JavaScript code and runs in the context of the web application.

During the setup stage, the in-browser runtime automatically attaches listeners and hooks to the web DOM objects that the developer wants to map into the AR environment. Listeners and hooks operate in pairs. They are executed when the corresponding DOM elements are created, deleted, or updated (from the server). The in-browser runtime also automatically serializes DOM objects as Alpaca Objects and creates those on the Alpaca Server through the RESTful API. The in-browser runtime also participates in creating the relevant event streams.

During the operational stage, the in-browser runtime is an asynchronous, event-based program that responds to changes in the DOM or event notifications received from the server. The concurrency management of the JavaScript functions and DOM functionalities are entirely left to the web browser. The in-browser runtime is aware of the server, but not aware of whether the scene on the AR device exists, because the user may not have elected to use an AR device. The in-browser runtime is also not aware of how the AR scene looks. Handling and rendering of the Alpaca objects on the AR device are left to the on-device runtime.

On-Device Runtime. For ease of development and feature parity between the different AR devices, the Alpaca On-Device Runtime (ODR) needs to be able to run on all supported devices, currently the iPhone X and Google Pixel. JavaScript is a common language between the two devices, which also aids in the interoperability with the Alpaca Server. The In-Browser Runtime is written in HTML5 and JavaScript.

On the iPhone X and Google Pixel, the registration support comes from Google's WebAR project. Internally, these use OS-provided libraries on top of Google's Chrome browser.

The on-device runtime is application agnostic. It simply queries and consumes Alpaca objects stored in the on-server object store. Strict interpretation and laying out of the scene, and rendering of the objects in the scene are all controlled by the on-device runtime, however. The same as in the in-browser runtime, all on-device objects have listeners and hooks attached to each object as well.

Rendering within the on-device runtime is quite efficient because of the design choice to prioritize the use of image-based rendering on the AR device. Each object appears as a simple texture mapped surface. The actual content of the image to be used as the texture is saved on the Alpaca Server in the object store. The rendering functions of the on-device runtime use `THREE.AR.js` for scene registration and THREE.js for rendering.

Image-based objects can be placed "on-surface," for example to mimic the appearance of being laid flat on top of a desk surface. The image-based object can also be placed directly facing the viewer, in that case, by interactively updating the image depending on the view angle change, the user can see a responsive 3D object in the AR scene. By limiting object complexities, the rendering performance is able to be further optimized.

Non-Invasive Event Handling. Having described the runtimes, let us now discuss the Alpaca Server from an operational perspective. During initialization (i.e. when a user invokes the AR extension for the first time), the Alpaca Server holds an empty scene. The application developer updates the scene through code in the browser extension based on the DOM element state of the application. To react to events from the On-Device Runtime, the developer can add listeners using the Alpaca API.

There are three kinds of listeners in the extension: the DOM listeners update based on the changing state of the host application; the event listeners update when the user interacts with the AR device; and the scene listeners allow the AR device to react to new scene graphs. From a developer point of view, these are grouped as DOM/Event listeners vs Scene/Event listeners.

The application developer writes code similar to Figure 4.3 (for Google Books). In (1), the extension creates a DOM listener that will react to changes in the application state. Each time the page updates, (2) the developer needs to get the latest images of the pages of the book. For each page, (3) a callback function reacts to press events from the On-Device Runtime by changing the current page in the web application. Finally, once the scene graph is complete, (4) scene on the AR device is updated.

After initialization, a scene should have been created. The developer can then set event listeners (e.g. press events) onto objects of this scene. Once these are set, the scene is pushed to the Alpaca Server. This process is illustrated in Figure 4.4 and is repeated each time the DOM listeners are triggered. This example is written so it recreates the entire scene from scratch due to the need to handle highly dynamic DOM changes in a reliable way.

4.3 Alpaca: Applications & Use-Cases

As coined in [110], "an interaction is an action by a user with an intent to change the state of an application". Among *select*, *explore*, *reconfigure*, *encode*, *abstract/elaborate*, *filter*, and *connect*, some of these interactions are better suited as spatial-oriented while others are better as surface-oriented. The best fit of the interaction techniques on each platform will be task dependent.

Abstract tasks [27] that users handle on-device or on-desktop can also vary. Search tasks such as *lookup* may be better suited on desktop, and *browse* better suited for AR. Search tasks such as *locate* and *explore* could fit on both environments. Query tasks such as *identify*, *compare*, and *summarize* can fit on both environments.

Alpaca's interaction mechanism allows application developers to make flexible design choices on interaction. All application code is in JavaScript, which is easy to prototype

```
Alpaca.listenDOM(domElement, function() { // (1)
  let domImages = domElement.getImages(); // (2)
  let ARScene = new Group();
  for (domImage of domImages) {
    let texture = loadTexture(image.src);
    let object = new TextureMappedPlane(texture);
    object.position = new Vector3(x, y, z);
    Alpaca.listenEvent("press", googleBooksNextPage); // (3)
    ARScene.add(object);
  }
  Alpaca.updateScene(ARScene); // (4)
});
```

Figure 4.3: A simplified code snippet showing the DOM and event listener setup process of the Google Books application; the elements are images containing the contents of each page of the book. In this example, the code is written to change to the next page on press. In the real application, there are separate buttons that go to the previous and next pages.

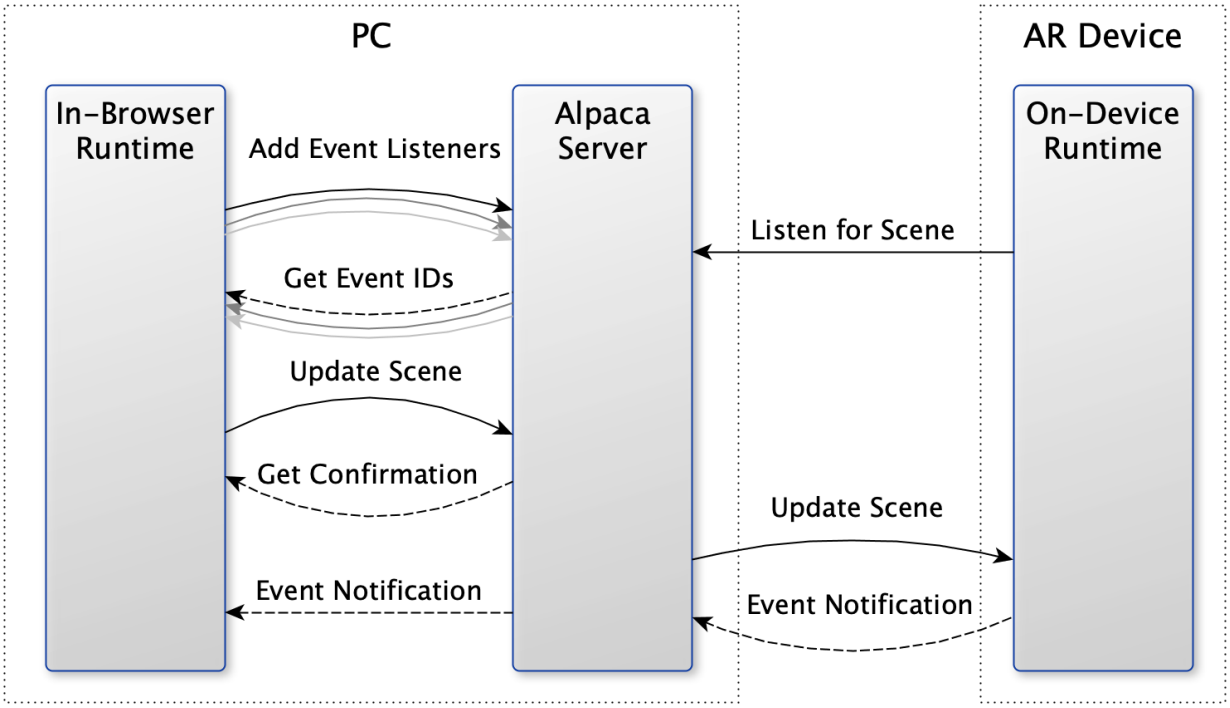

Figure 4.4: The scene and event listener update process when there is a DOM change in the web application, and a sample event notification flow upon user interaction. Arrow represent HTTP requests; curved arrows represent pairs of requests and responses.

and iterate. All the application code runs on-desktop, in the typical web application setting. This is the most native programming environment for many of today's application developers [w3t].

The application code is triggered on first load, its main function is to create the AR scene, register the scene with the server, together with all the objects. The application developer does not have to worry about setting up listeners, hooks, or event streams. The desktop runtime will set those up automatically. Each object is then annotated as to whether it responds to different kinds of interaction events.

Last, after the Alpaca objects have been created and properly tagged, the application code assembles the scene. The simplest way is to just add all objects into the same list. However, the application developer has the complete freedom to generate new objects, ones that are not in the DOM at all.

The assembled scene and associated objects are serialized and submitted to the Alpaca Server as one JSON object. The server will parse through the JSON, register objects individually, and store them. Scene objects are just another object of peer status as all other Alpaca objects. Again, as shown in Figure 4.2, the on-server object store is general and object-type agnostic and simply answers queries by the URI of the objects.

In this section, three application scenarios using Alpaca are shown. The first application extends a common website used by many people (Google Books), and expands the user's workspace to include books in the real world (Section 4.3.1). The second example looks at an educational webpage with content that can inherently benefit from 3D exposure (Section 4.3.2). The last example illustrates the creation of an AR extension when the developer has complete control over the application source and server (Section 4.3.3). Performance results for the applications are detailed in Section 4.4.

4.3.1 Google Books

Google Books is a large collection of digital books, which has scanned over 25 million books [Heyman]. Using Google Books, having multiple books open and scanning through them is now a process of switching between browser tabs and viewing one at a time, rather than having the actual physical books open on a desk.

The Google Books Alpaca application augments Google Books' resources with a digital, spatially-extended workspace. The user can continue to use the web interface for doing the initial research, but they can also pull out snippets or entire books to sit around them in their workspace for easier referencing.

Book to Plane Mapping. Google Books renders books using images of each page as simple HTML `<img>` tags. The user-written script takes these images and uses them as textures for THREE.js plane objects. It then uses Alpaca's `updateScene` function to send the object to AR.

Google Books server restricts the access of their images to users on their website (e.g. via the browser). Due to this reason, the Alpaca application must explicitly upload the content to the Alpaca Server; more data than just the new scene and interactions. Hence, this application is considered *asset-heavy*.

Figure 4.5-Left shows a user viewing two books in AR through their phone, while making notes on their laptop. Figure 4.5-Right shows a closeup view of one of the books in front of the user within AR. The arrow-shaped buttons on the left and right of the book let the user go to the previous and next pages respectively.

Interacting with Books in AR. For interacting with the books in AR, the script creates two 3D arrow shapes using THREE.js and uses Alpaca's `addEventListener` function to add a click event for the arrows. In the callback function provided to `addEventListener`, it triggers the Google Books code as if the user switched to the next page in the browser.

4.3.2 Exploration of Biodiversity Through Maps

Maps are abundant on the web, and increasingly help people navigate and better understand their surroundings at different levels of granularity. In most cases, the maps are projections of a 3D world on to a 2D plane. In other words, they inherently contain 3D information. With a simple mapping provided by the developer, a 2D map on a website can be viewed in 3D in AR on the user's desk. This use case is centered on the Species Mapper web application of the National Park Service, which visualizes and compares species distributions across the Great Smoky Mountains National Park.

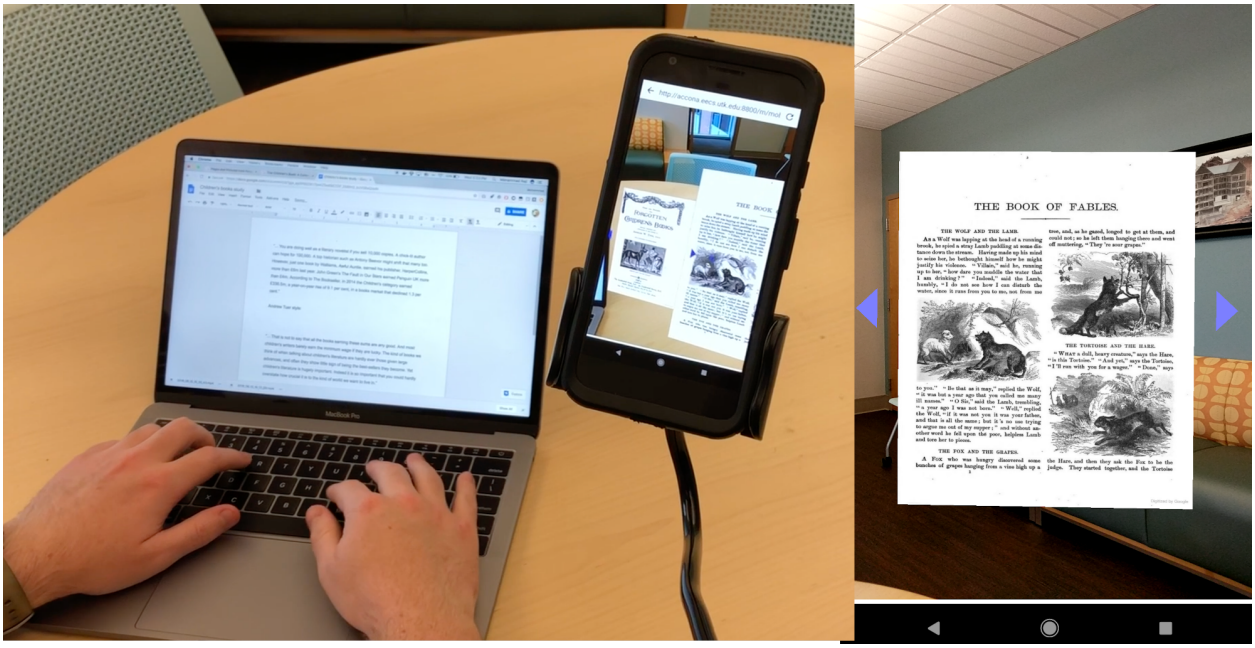

Figure 4.5: (Left) The 3rd person view of the Google Books application shows how the user can interact with their laptop and take notes based on the books that they have open in their AR workspace. (Right) The close-up view of the AR device shows the level of detail capable on the Google Pixel.

Elevation to Height Mapping. The *Species Mapper* web application renders a map of the Smoky Mountains and provides panning and zooming capabilities. Additionally, users can view the distribution of different species rendered as an overlay on the map. Internally, the web application uses *leaflet.js*, a de facto library for GeoMap-based applications. Leaflet renders maps using tiles of images within the DOM.

The application starts by parsing the DOM for any underlying maps, then create a THREE.JS plane object with two textures. One texture is used for the picture of the map, another is used to encode elevation. The information from these are both gathered from Species Mapper. To make the map 3D, the user script uses a vertex-shader to map elevation to height. Finally, the THREE.js object is given to Alpaca's `updateScene` function that renders the AR version of the scene on the connected AR device. The result is shown in Figure 4.6.

The Species Mapper application is *scene-heavy*, because the vertex shader used for height mapping introduces more complexity to manage the scene and requires more complex usage of the AR device's GPU. Even though such complexity may hamper this application's portability to a wider range of devices, that complexity has enhanced user experience rather significantly.

The map images of Species Mapper are cloud-hosted and easily usable by the AR device. This is because Alpaca also uses URIs as the universal identifier of assets as well. In other words, the images are not uploaded to the Alpaca Server for hosting and only the URIs are sent to be fetched by the AR device through the Internet.

Map Interactions. After the user selects a species from the host web application, the desktop runtime is notified that the DOM has changed and it begins to update the THREE.js object based on the map. The AR device then updates the scene. Users then interact with the model.

4.3.3 Exploration of Online Recommendations

There are many web applications that include a powerful online recommendation engine. YouTube is a prime example of that. The focus of this application is to develop a new

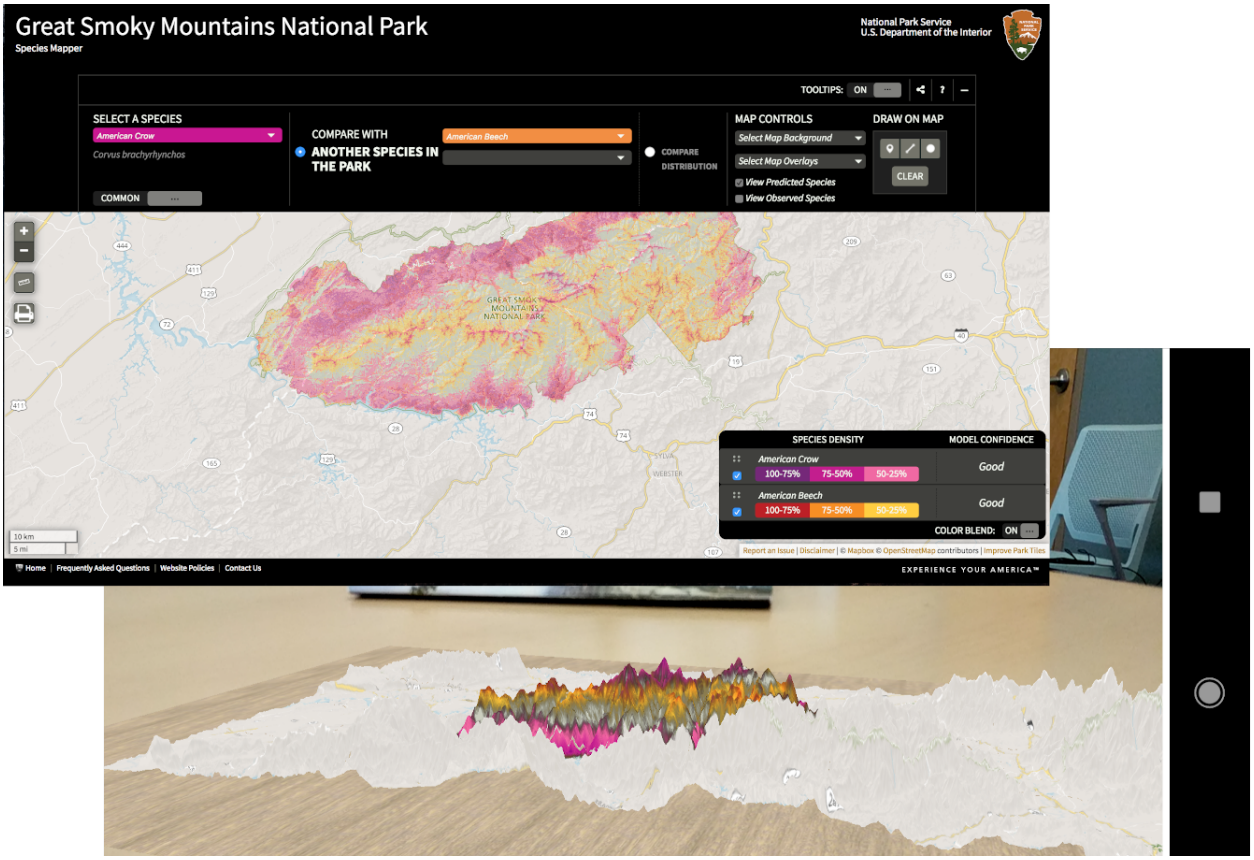

Figure 4.6: The AR view of *Species Mapper*. The elevation of the mountains is mapped to 3D height by the DOM-Scene mapper.

YouTube-based application using Alpaca that allows users to interact with and explore in both the 2D screen and 3D physical space.

In today's web apps, recommendation is usually limited to a single 1-dimensional list of suggestions. A user can select one item from the list, upon the selection, a new 1D list of recommendations are computed based on the selection and delivered to the user. At any particular time, the user sees one list of recommendations. In a large part, this is due to the limited space on a 2D screen, and that selecting from a 1D list is intuitive and direct.

The application builds on automation of the above process using YouTube's public API. Specifically, after retrieving the recommendation list for one video, for each video on the list, a script is used to automatically retrieve the recommendation lists for those videos. This process recurses k-levels until a threshold number of videos are retrieved.

The process creates a directed graph. This graph is not a tree, because the same video may appear in the recommendation lists obtained during different levels of recursion. Each node in the graph is a video. To be performant in rendering, typical graphs are capped at 100 videos which equates to $k = 2$ levels of recursion when accounting for 10 related videos fetched from YouTube per video. Edges between videos indicate what YouTube's recommendation engine considers as related: i.e. a connection from A→B means that B is in the list of videos related to A.

The script automatically presents the video recommendation graph in the DOM. Using Alpaca, the graph can be easily made to appear on the AR device in 3D. The AR view is comprised of the set of nodes in the graph, represented by their video thumbnail images. These thumbnails are accompanied by interactive buttons that either "like" or "dislike" the video, which are then used by Support Vector Machines (SVM) to refine the presentation of the graph.

Figure 4.7 shows a typical workflow for the mixed-space YouTube application. First the user starts the desktop application and is prompted to use their AR device. The user can now intuitively switch between global and local views of the suggested videos. For better visibility and exploration, the AR runtime will automatically orient the thumbnails so they always face the user.

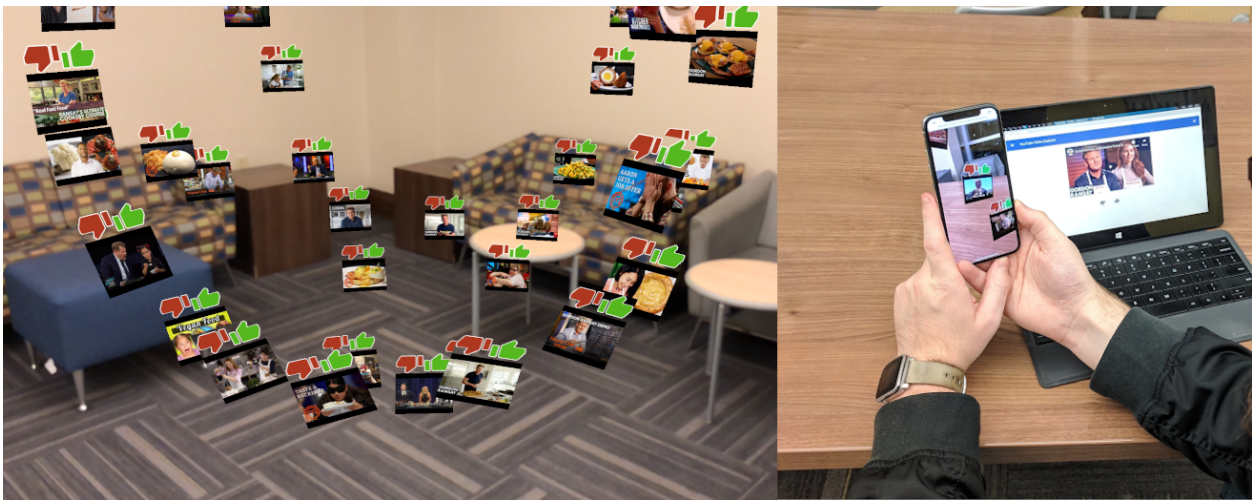

Figure 4.7: (left) The AR view of the recommended videos layout shows how different videos fall into distinct groups. (right) The third person view shows what the user sees on their AR device and on their laptop.

A user can interact with the AR video recommendation graph by simply tapping on a video, the desktop in-browser view updates in a synchronized way. Event listeners are transparently attached to both the videos in the DOM and the videos in the AR scene graph. These listeners allow the user to make changes in either the desktop or AR views and have these changes affect the application as a whole.

Due to the number of videos available to the application and the difficulty in navigating such a graph, an SVM aids the user in filtering the collection of videos. This filtering occurs by the user selecting videos they "like" or "dislike" through in-device tap interactions. After a few selections, the user can trigger a classify function, upon which a new SVM is trained and tested using the tags of the videos. This acts as a rudimentary way to increase the dimensionality of each video and improve the quality of predictions. The filtering all happens locally in the browser.

This application is *update-heavy* because the scene graph has to be rebuilt every time the user filters the graph or watches a video and hence triggers the recommendation list to be refreshed. These updates and the resulting reconstruction process could become a bottleneck, strain the AR device, and hamper rendering rates.

4.4 Alpaca: Scalability Study

The three demo apps in the previous section also serve as test cases to better understand the performance characteristics to bridge web with AR. In all tests, the Alpaca Server and the browser are co-located on the same PC — a ThinkPad T420 laptop with a 2.80GHz Intel Core i7-2640M CPU and 8GB of RAM. The focus of these tests is on the following metrics.

AR startup time should be $< 5s$ from when the desktop application creates the scene to when it's visible on the mobile device. This roughly corresponds to the time it takes for a user to pick up their mobile device after using their mouse or keyboard to interact with the desktop application.

AR frame rate should be in the $30 - 60$ FPS range to be usable. The Alpaca runtimes are not directly in control of this metric because it is directly tied to the performance of the AR device, however due to rendering on the AR device instead of a remote-rendering

technique, this frame rate was found to be achievable. It is worth noting that the frame rate is affected by the rendering time as well as overheads due to AR registration and sensing. Thus, AR frame rate can only improve so much before the limitations of the AR device are reached.

Server memory footprint should be < 100 MB to limit the effect the Alpaca system has on other services on the user's laptop. This range is around that of a normal web app and can be seen as unintrusive.

Total extension code should be around $100-1000$ lines, with around 50 lines of Alpaca application code. The Alpaca system is reusable and complex, but the application code using the API is kept simple and in only a few additional lines of code. Most THREE.js applications are already around the 1000 line mark, so adding 50 lines of Alpaca code is relatively minimal.

Alpaca Server Throughput. Next, the data transfer rates are evaluated. These rates directly impacts the speed that the desktop extension can update the scene graph. On average, a scene is on the order of kilobytes, but the assets may total 10s of megabytes. This means that supporting 100s of MB/second should be sufficient for Alpaca applications.

Additionally, there are two test cases of importance: serial and parallel uploading. Serial uploading is where each MB is uploaded after the previous one completes. In parallel uploading, 4 different objects are sent at the same time, like a browser would do in an asset-heavy application.

For the test, a script repeatedly uploads blocks of 1 MB (around the size of a single image) and measures the total time required to upload all this data. Then, the total data size is divided by the time to get a rate in MB/s. This trial is repeated in both serial and parallel modes. Results are shown in Figure 4.8.

Due to the asynchronous way the server was written, the results indicate a higher upload rate during parallel uploading than in serial. It is likely that the much higher increase for parallel uploading of 512 MB is due to matching the operating system's buffer sizes without exhausting lower level caches. Overall, it is evident that the Alpaca Server is able to handle a sustained upload rate of 400 MB/s which is more than enough for the applications tested.

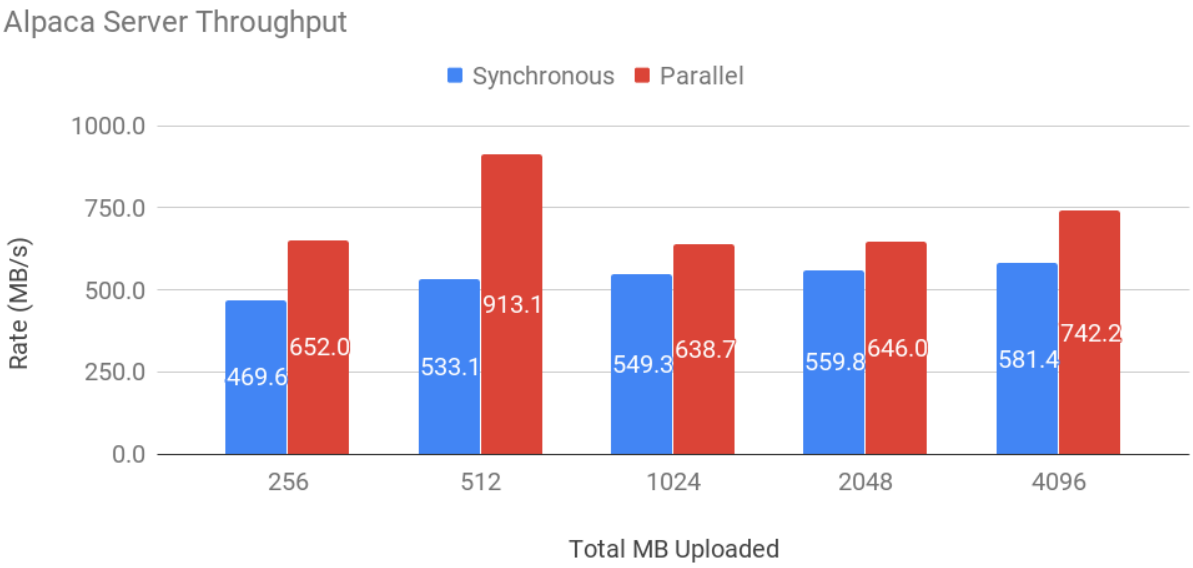

Figure 4.8: Server throughput as measured in synchronous and parallel modes. The data shows the maximum upload rate that can be expected of the system. The synchronous mode corresponds to scene- and update-heavy applications, while the parallel mode relates to asset-heavy applications. These are differentiated by the manner of uploading, between those that must happen in sequence and those that can be concurrent.

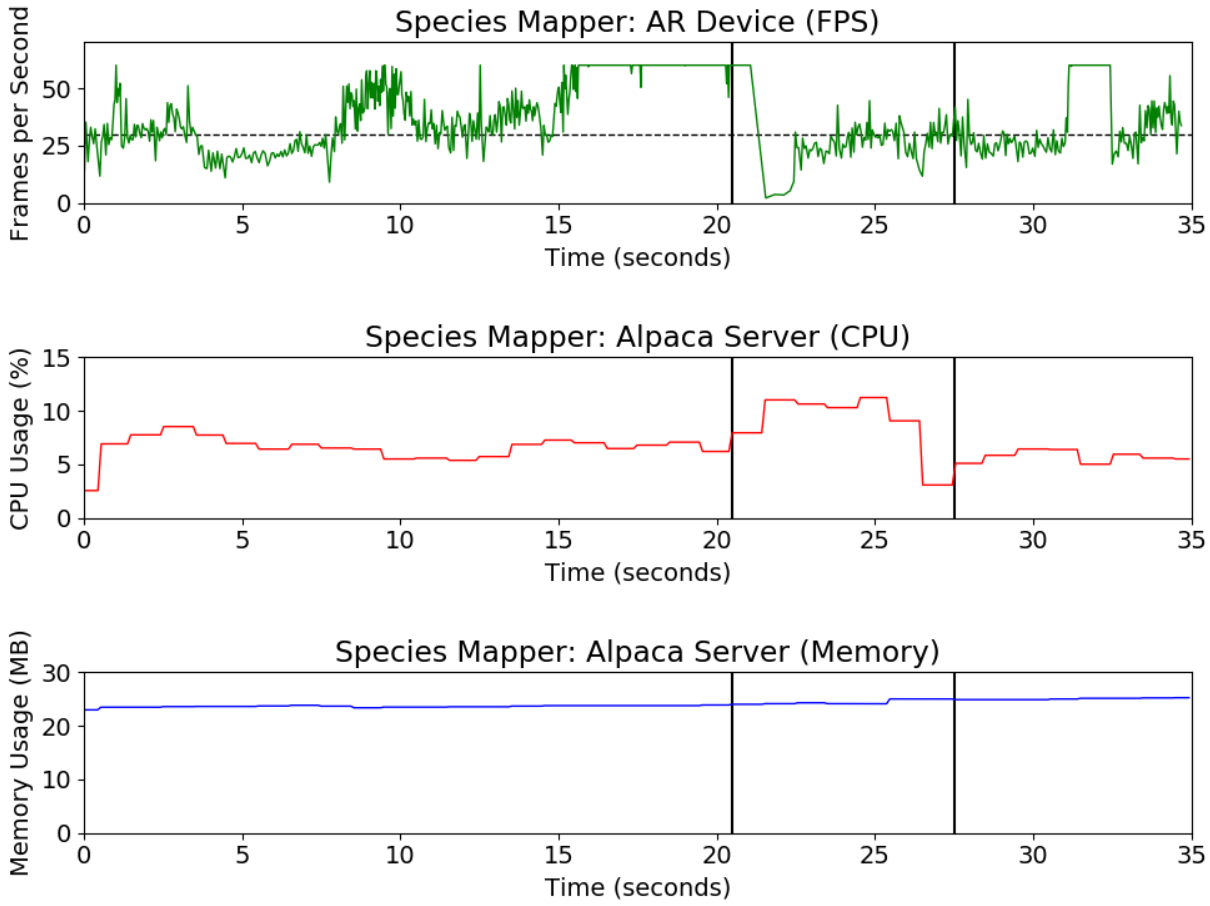

Figure 4.9: Alpaca Server and On-Device Runtime performance for the Species Mapper app. This is an example of a scene-heavy application that requires more compute resources to render the AR application, which lends to lower-than-average frame rates due to the AR device used. At the first mark in the graph, the user selected another species layer. The effect of this change is finished around the second mark.

Application Sample Runs. In this section, the goal is to test whether the Alpaca server is general and efficient enough to support multiple applications that use it, regardless of whether those applications are asset-, update-, or scene-heavy. This is evaluated through a series of sample runs for the Species Mapper and Google Books apps. In those two apps, the AR objects, and the related geometries, are auto-extracted from the existing web applications. The YouTube example app is not ideal for use in this performance testing, because that AR app required us to programmatically crawl YouTube recommendation listings and then custom create the AR objects after the multi-step crawl. The resulting runtime overhead includes multiple rounds of YouTube API queries, which is more complex. The collected metrics include: the render time for each frame of the AR device, the CPU usage by the server, and the memory usage by the server. All tests were performed with an iPhone X running iOS 11.1.2 to eliminate as much AR device latency as possible to evaluate the server more accurately.

The render time is measured in milliseconds and measures the time for each frame from when the rendering started to when it finished. It experiences peaks and troughs based on what the AR device is doing at the time. For example, when processing a new scene, the AR device is expected to have a longer render time and consequently a lower frame rate. The results are shown in frames per second (FPS), instead of rendering time.

The CPU usage is measured in terms of percents of a core, as measured by the `ps` utility on Linux. This percentage can be greater than 100% when it uses multiple cores; however, because of the single threaded nature of the Alpaca Server, this was never to be the case with the applications tested.

Memory footprint is measured in MB and corresponds to the virtual memory usage reported by `ps`. The Alpaca Server will use more memory as more objects are added into the object store, though it was found that this increase is proportional with the amount of data it is managing. The results are in Figures 4.9 and 4.10.

Figure 4.9 shows data from the Species Mapper app, which is scene-heavy. The render rate is between 10-30 FPS. Between the two marks on the chart (Figure 4.9), the user added a species layer and triggered a scene update on the AR device.

Figure 4.10 shows the Google Books sample run, which is an asset-heavy application. The results show that it demands higher CPU and memory usage by the Alpaca Server to handle these assets. Between the two marks on the chart (Figure 4.10), the user was scrolling through different pages of the book, thus triggering multiple updates to the assets in the Alpaca Server. Once this interaction has completed, the Alpaca Server's usage drops back to original levels until the user interacts with the desktop runtime again.

A summary of frame rates is shown in Table 4.1 (Left). Although a proper cross-device performance analysis is out of scope for this paper, rudimentary testing of these applications show that the Google Pixel can achieve an AR frame rate of $10 - 30$ FPS consistently, while the iPhone X is able to consistently output at $30 - 60$ FPS. It is likely that a major reason for this difference is that the iPhone X performs AR registration and sensing closer to the hardware.

Code Size Evaluation. As one of Alpaca's key goals, it is intended that developers can make use of its infrastructure in only a few lines of code, the majority of which is typical THREE.js code. To this end, a final result was measured: the size of each application (shown in Table 4.1 (Right)).

The total lines of code is measured as the number of lines, including comments and empty lines, in the extension JavaScript file. Of those lines, the "Alpaca related" lines are those that directly interact with the Alpaca API (i.e. calls to `listenDOM`, `listenEvent`, `updateScene`, and functions that get passed to event listeners). It was found that each of the applications is in the range of typical THREE.js applications in terms of size, and that the percentage of Alpaca related lines is below 15% in each case.

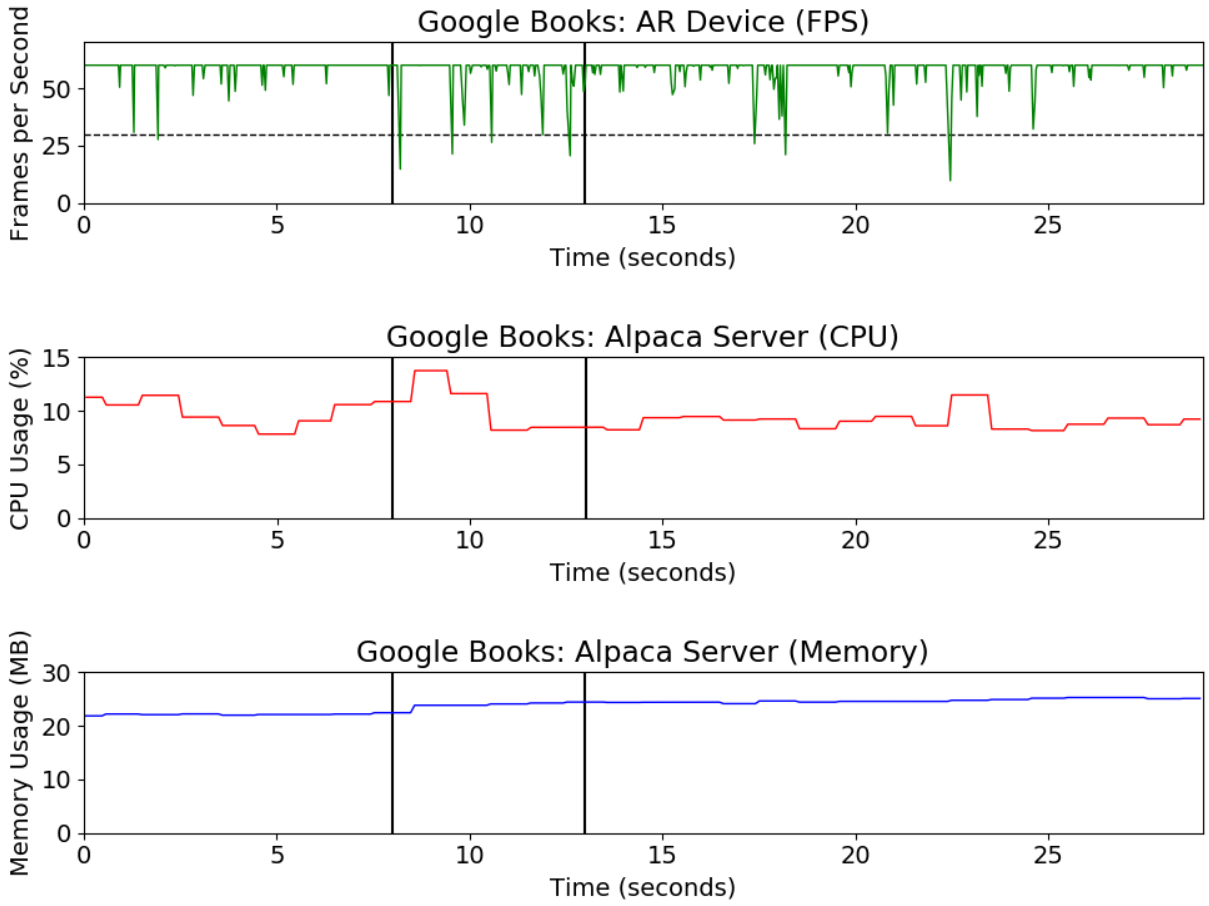

Figure 4.10: Alpaca Server and On-Device Runtime performance for the Google Books app. This is an asset-heavy application and must upload assets directly to the Alpaca Server to get around access restrictions imposed by Google Books. Between the two marks on the graph, the user was scrolling through the book, triggering multiple scene updates, and consequently, multiple asset uploads.

Table 4.1: The AR frame rate and lines of code of each application. The frame rate is unsteady during scene changes but then levels out afterwards. In all cases, the total lines of code is less than 1000 and the Alpaca contribution is less than 50 lines of code, under 15% of the total lines of code.

	FPS	Code Ratio
Google Books	$16 - 60$	26/198 (13.8%)
Species Mapper	$4 - 60$	17/397 (4.28%)
YouTube	$30 - 60$	18/466 (3.86%)

Chapter 5

Hypothesis-Driven Service

5.1 Overview

In this chapter, I study the design and implementation of a hypothesis-oriented VaaS. This kind of VaaS is unique in that it does not have pre-defined visualizations that the user is choosing from, and is instead almost entirely user-directed. Specifically, the VaaS in this chapter uses small and self-contained programs as the format for requests. It is this idea of treating VaaS as an engine for dynamic execution that is widely applicable to hypothesis-oriented visualization as a whole.

Graph is a universal representation of complex relationships and a data structure common in many application domains. A widespread ability to understand the complex relationships embedded within those big graphs will enable and accelerate discoveries.

In this chapter, I present a VaaS to render large graphs. The system is called the Graph Shader Engine (GSE) based on a new concept I refer to as a "Graph Shader."

Graph Shaders revolve around a domain specific language called the Graph Shader Language (GSL). The design of GSL is intended to provide a reusable and efficient interface to give users an expressive means to control graph renderings. Typically, interacting with a graph means to pan, zoom, pick, or brush using a pointing device like a mouse or trackpad. I believe Graph Shaders can provide an additional means of exploration.

GSL is inspired by how GLSL shaders have enabled OpenGL renderings to become more creative and expressive. From this respect, I've designed GS shaders to help users to express their exploratory hypotheses more effectively using graph primitives. Corresponding to vertex, geometry and fragment shaders that are available in GLSL shaders, GSL also has three components: (i) positional, (ii) relational, and (iii) appearance shaders. The GSE parses GSL and automatically generates the corresponding GLSL shaders in a transpiling process. In this way, any graphics hardware that supports current OpenGL standard will be able to support graph renderings that use GS.

Graph shaders also draw inspiration from the design of OpenMP programs and their use of "`#pragma`" statements to interleave compiler directives within ordinary lines of code. Using `pragma` directives, programs in GSL can opt in to restrictive data access patterns to improve their performance. The combination of code in GSL and `pragma` statements enables the code to become a standalone GS Program that can be executed by the GSE. Graph data in the GSE is made available to GSL using OpenGL's Shader Storage Buffer Objects (SSBOs), allowing for simplistic and powerful data access patterns.

I use three driving applications to demonstrate analytical benefits of GSE. Each application targets a particular dataset, with sizes ranging from 1.2 to 3.8 million nodes and 6.2 to 63.5 million edges. In all cases, the driving application falls under the use scenario where the graph that contains large and complex relationships will continue to evolve over time, and it's beneficial to host the most up to date version of the data as a cloud resource for remote users to share and use. The specific example use cases are:

- To identify the evolving interdependency networks among open-source projects using the JS-Deps dataset of JavaScript Dependencies among NPM packages [npm] (shown in Figure 5.3).

- To reveal conversation patterns and relationships among users on an online platform, such as Stack Overflow, using the SO-Answers dataset from the SNAP project [67] (shown in Figure 5.4).

- To track the citation networks between patent applications of US Patent and Trademark Office using the NBER-Patents dataset [44] (shown in Figure 5.5).

The contributions of this chapter are: (i) to develop a graph shader model that includes positional, relational and appearance shaders, (ii) to show that GS shaders can be transpiled into GLSL shaders, and (iii) to demonstrate the analytical power of GS shaders in three different application settings. Like in the setting of OpenGL shader playgrounds, GS graph shaders can be easily stored, version controlled, and shared as well.

The remainder of this chapter is organized as follows. I start with Section 5.2 discussing the development of the GS Engine and the GS Language. Section 5.3 showcases three applications for the visualization and analysis of large graph data. The results in Section 5.4 show the efficiency of my system.

5.2 Graph Shaders: Design

In this section, I describe the design and interface of the GS, comprised of three nested components. The GS Service (Section 5.2.1) is a distributed set of web servers that serve an HTTP interface to the GS Engine. The GS Engine (Section 5.2.2) is a C++ executable that uses OpenGL to run code written in the GS Language. The GS Language (Section 5.2.3) is an extension of the OpenGL Shading Language (GLSL) to control graph processing and rendering.

5.2.1 GS Rendering Service

To support the efficient visualization of graphs using Graph Shaders, the GS Service is a VaaS designed as a portable and optimized runtime for data management and OpenGL state management. The core of the GS Service is a renderer called the GS Engine (Section 5.2.2), responsible for all of the data management and rendering needs. The GS Service makes this core functionality accessible as an HTTP interface, ensuring generality to web browser clients and any program with support for HTTP requests. The use of HTTP as the protocol is the only viable choice for general cloud systems and has been shown to be efficient and scalable in previous VaaS [90, 86, 50].

Flexible Deployment. The GS Service makes use of the containerized deployment methods previously found to be lightweight and reliable [90, 86, 50]. The Docker [5] instance

containing the server side can be deployed anywhere, including a standard desktop computer or rack mounted computing nodes in typical research environments. With many-core CPUs having become widely available, software based OpenGL implementations can be used. However, all results reported in this work are from using actual GPUs. Unique to GS, GPU is central to delivering high performance. Context switching on GPU is very expensive, and a hard barrier to circumvent. In result, when provisioning the GS Service, each instance requires access to a separate GPU.

It's worth noting that a Docker instance of the GS Service requires only incremental resource in comparison with the resources needed to manage the graph data. When a target dataset fits onto a user's laptop, the service can be deployed directly onto that laptop as well. In result, at an application developers discretion, the GS instance can be deployed as a companion process that runs on the same desktop as the application, or run on a lab oriented cluster, or on an off-premise managed cloud. All of such complexities can stay completely transparent to the application's users. With such a general method of deployment, users can interactively explore the data using the same computer that hosts the data, or other laptops and tablets owned by the users.

Hourglass API Access. Dating from nascent days of Unix to recent scalable systems, such as MapReduce [31], a plethora of research has shown that a restrictive, narrow API design actually enables system scalability [19]. This common design is known as the Hourglass Model. The hourglass model has been shown to be effective in prior works of cloud-based visualization [90, 86, 50]. Specifically, to allow rendering requests as the only kind of access to a cloud-based rendering service. In addition, the rendering requests are completely independent of each other, and hence can be processed in parallel on elastic resources.

Tile-Based Task Parallelism. For interactive use of a VaaS, there is a need to minimize the time between user interactions and visual responses. The GS Service embraces this idea through task-based parallelism of individual tile requests. A rendering request to the GS Service is concerned with only an image tile. The tile-based rendering requests are processed independent of each other. Supposing that the final image needs to be 1024×1024 and that the image tiles used are sized 256×256, there are a total of 16 tiles to rendering. Applications

should be able to flexible manage rendering requests. For example, modern web browsers such as Chrome and Firefox can transparently manage streams of web requests with fault tolerance. A container based architecture can elastically manage a batch of requests and process them in parallel.

Handling Rendering Requests. A rendering request encodes the desired visualization characteristics in a declarative, stateless way. The GS Service processes rendering requests using the `Flask` library for Python, transforming a normal Python process into a custom HTTP web server. Requests are parsed in parallel on each instance of the service and are put into a queue for rendering. A single OpenGL thread performs all of the rendering tasks in serial for efficiency in managing the OpenGL context. Responses to the application are handled in parallel. Due to the nature of how HTTP requests are generated from a web browser, many requests will be queued up at once, allowing the GS Service to parse requests concurrently further optimizing the serial rendering pipeline.

Responding to Rendered Requests. The response to a rendering request is an encoded rendered image. GS encodes images using either the JPEG or PNG formats. Due to the ubiquity of hardware accelerated encoding and decoding of JPEG images, the GS Service defaults to using JPEG for responses. For a web application, stitching together JPEG images tiles can be done transparently by the web browser.

5.2.2 GS Rendering Engine

The GS Engine is the executable that directly manages the OpenGL rendering context. Its responsibilities are restricted in this way following the commonly accepted principles of separation-of-concerns in software development. The partition between the HTTP interface management and the OpenGL content management serves to simplify their respective implementations. This section describes the design and implementation of the GS Engine that executes code written in the GS Language.

An obvious alternative design to executing GS Language code on a GPU is to forego OpenGL entirely and use GPGPU interfaces. This approach would provide maximal flexibility, but the goal for GS to be a visualization system primarily would mean the GS Engine would have to re-implement the rendering pipeline built into OpenGL. I have found

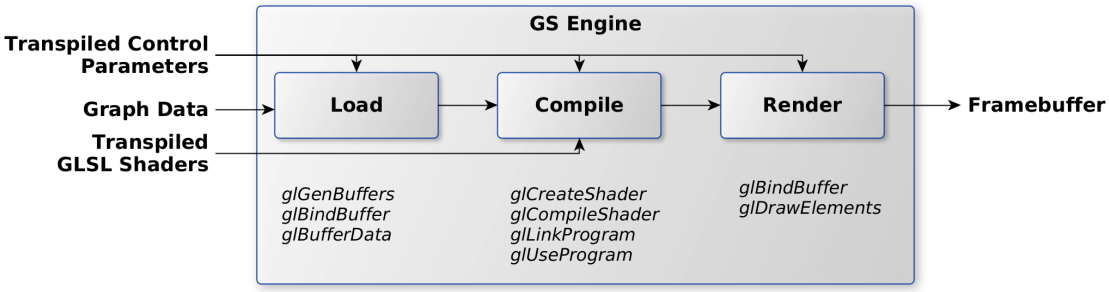

Figure 5.1: Overview of GS Engine, the experimental system to use GS graph shaders to render graphs in OpenGL. The most relevant OpenGL calls in each stage are listed accordingly.

that the design of GS Language shaders benefits from the familiarity of reusing the OpenGL rendering pipeline.

Figure 5.1 illustrates connection between GS Language shaders and OpenGL shaders. After the GS shaders have already been created, the first execution step is for the GS Engine to load graph data into GPU memory.

After evaluating options such as Vertex Array Objects (VAOs) or Vertex Buffer Object (VBOs), the best choice for GS is to use SSBO (Shader Storage Buffer Objects) to hold graph data in OpenGL. SSBOs are especially suitable for use in OpenGL shader code because their interface is a simple typed memory array capable of random access. In the GS Language, the attributes of each SSBO are configured using `pragma` directives inline with the shader code. This design in essence enables us to use a wide range of arrays and make those globally accessible by all GPU threads.

The efficiency of GPU memory access then depends on the GPU's memory caching mechanisms, for which the tests have shown satisfactory results. From this respect, the global accessibility makes each GS shader simpler and more succinct. During rendering stage, the render call can also treat the SSBO data as elements array (interleaved source and target node indices). Due not needing different ways to handle data in different stages of the pipeline, GS shaders in the end are just GLSL code with `pragma` directives. The whole GS Engine is written in C++.

5.2.3 GS Language (GSL)

The core design of the GS Language is to consider the simultaneous pairing of graph processing and analytics with graph rendering. In this manner, it is helpful to reference existing graph analytics task taxonomies in the literature as these provide a framework for the variety of analytics tasks possible with Graph Shaders and its situation within the literature. Specifically, I consider the static graph analytical task taxonomy by Lee et al [65] in 2006, the Andrienko Task Framework (ATF) [12] for temporal graph in 2006, and more recent extensions by Aigner et al [10] in 2011 and by Kerracher et al [61] in 2015.

Overall, the known set of tasks is already comprehensive. For example, for static graphs, Lee et al summarized five categories [65]: (i) topology-based tasks, (ii) attribute-based tasks, (iii) browsing tasks, (iv) overview tasks, and (v) high-level tasks. For temporal graphs, the ATF framework [12] breaks down user needs as (i) lookup, (ii) comparison, and (iii) relation seeking. After the ATF framework has been extended [61], many more sub categories have been identified, with the top two categories of tasks being: (i) attribute based tasks, which can be subdivided further according to whether the task is on values or sets, and (ii) structural based tasks, which can be subdivided further according to whether the task concerns structural patterns or relations between elements.

The primary objective of Graph Shaders is to render useful raster framebuffers with flexible data processing. From this respect, the choice of features for graph rendering is based on intrinsics in the data, specifically: (i) attribute based features, (ii) structural based features, and (iii) topology based features.

Graph Shaders can directly support attribute based features and help identify relations. Indirectly, when a user visually inspects and manually controls the graph shaders, the renderings can help users visually comprehend structural based features as well. Shaders can handle topology information, but only after the topology analysis results have been created and provided by graph processing in a pre-computed manner.

Graph Shader's primary user benefit is to enable powerful visualization hypotheses in a more abstracted manner. The design of GS is as a graph-focused extension on OpenGL. For this reason, GS shaders are designed to have a direct mapping to OpenGL's GLSL shaders. Specifically, GS shaders have three components: **Positional Shader**, **Relational Shader**, and **Appearance Shader**, which map to the GLSL vertex, geometry and fragment shaders, respectively. The transpiling process from Graph Shaders to GLSL shaders is automatic. In result, GS shaders are designed to be as portable as GLSL shaders. The relationship between GS and GLSL shaders is illustrated in (Figure 5.2).

Lastly, in a related manner, while most graph layout algorithms operate in the 2D space, OpenGL operates in the 3D space. In this regard, if graph data can be ordered, for example, to distinguish foreground vs background, this information can be leveraged to improve the legibility of renderings. Even though this is not required for GS shaders to work, this simple

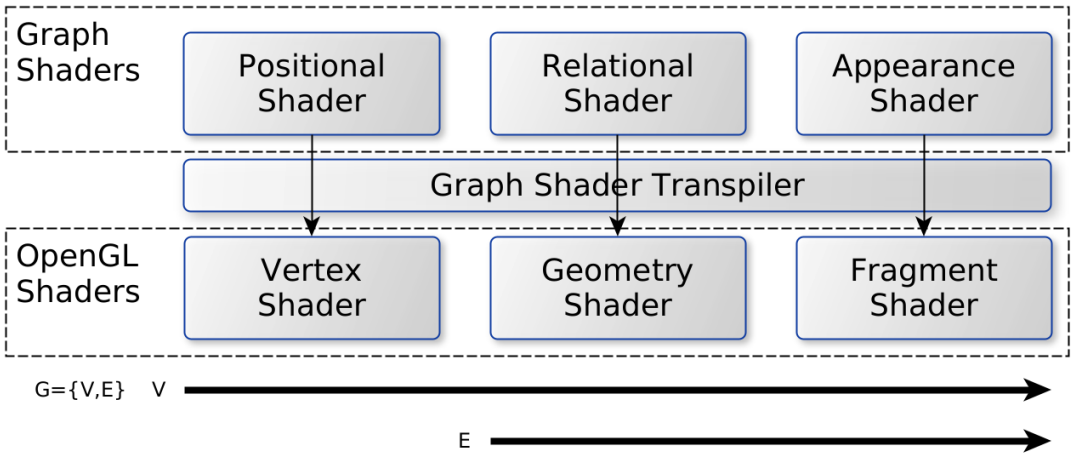

Figure 5.2: Overview of Graph Shader Language.

pre-computation step does help generate renderings that are more visually intelligible. This ordering operation can be thought of as creating a Z-depth. However, typically an explicit numeric Z value is not required, an implicit ordering can be sufficient.

GS Programs & Pragmas. Before the GS Engine operates on Graph Shader code, it is pre-processed by a Python program called the GS Transpiler. This program serves as a minimal wrapper to the GS Engine to minimize the amount and complexity of parsing code in C++. The GS Transpiler's role is to translate Graph Shader Program (extension "`.GSP`") scripts into regular OpenGL shader code and control parameters that instruct the GS Engine how to read data buffers, bind buffers to GLSL variables, and every other aspect of rendering graph data. GS Program code is instrumented with `#pragma` directives that configure the resulting control parameters used within the GS Engine. The pragma system is created with extensibility in mind and any number of pragmas could be coded up and have new engine code to support them.

The specific purpose of the `pragma` directives is to instruct the GS Engine on what buffers to generate, what data files to load, and what "scratch" buffers to allocate. Two pragma statements are necessary. First, the "`shader`" pragma allows a single GS Program to contain the interleaved source codes for each of the positional, relational, and appearance shaders. In the beginning, GLSL code is written to the "common" section which is treated as the preamble to each other shader.

The second required directive is the "`attribute`" pragma to declare the name, type, and size of graph data attributes. This directive interprets a standard array declaration in GLSL and configures the GS Engine to populate that array for the shader code to use. Each attribute is backed by an OpenGL SSBO. The size of the attribute can be a fixed number or the special constants `N` or `E` representing the number of nodes and number of edges, respectively. The attribute's name is reused for the on-disk path to read the data file.

The "`scratch`" directive functions the same as the `attribute` directive, except that the data is zero-initialized. The buffers allocated by this directive are intended as general purpose storage for the shader code to use for arbitrary computations. Unlike the `attribute` directive, `scratch` supports the OpenGL `atomic_uint` data type, also initialized to zero. Every atomic scalar variable is backed by a single OpenGL Atomic Counter Buffer, with each

corresponding to a different offset within that buffer. The atomic data type is also handled specially by the GS Engine because their final values are included alongside the rendered image, allowing for quantitative properties of the graph to coincide with the qualitative properties of the rendered image.

Data Flow. I will now illustrate how GS shaders set up data buffers for GLSL shaders to use. Listing 5.4 is a snippet of a very simple GS shader declaring a `float` array via the `attribute` pragma, and then retrieves a value from that array, and assigns that value to a built-in variable of the GS Positional shader. The array is automatically managed by the GS Engine as an OpenGL SSBO and its value is initialized from a file on disk. The OpenGL binding index for the SSBO is determined by the GS Transpiler.

Listing 5.5 represents the GS Engine code for the transpiled GLSL and the compiled C++ programs. The OpenGL binding index is read from the control parameters generated by the GS Transpiler. This index is used for both data loading on initialization and for data mapping during runtime.

Positional Shader

Example. The GS positional shader is the first shader in the pipeline. Listing 5.6 demonstrates a complete positional shader including the pragma directives for binding graph data. This shader is very simplistic because the graph layout was already computed externally and not modified inside the positional shader. This is also the exact positional shader used by the examples in Figure 5.3.

Usage. GS positional shaders differ in purpose from most OpenGL vertex shaders because the GS Engine uses SSBOs for all graph data. Normally, a vertex shader sets up variables to be forwarded or interpolated in later stages. That's not necessary in GS shaders because all graph data is accessible to all stages. In addition, positional shaders take on a more computational role by being the ideal place to create new graph node attributes and to initialize new graph edge attributes for further processing in later stages. Positional shaders are especially useful for computing aggregate information about the graph as a whole, such as computing the global lower and upper bounds of node attributes.

```
#pragma gs shader(common)
#pragma gs shader(positional)
#pragma gs shader(relational)
#pragma gs shader(appearance)
```

Listing 5.1: The `shader` pragma specifies in which shader to put the code that follows.

```
// Read floating point x coordinatex as node data
#pragma gs attribute(float XPosition[N])

// Read integer unix timestamps as edge data
#pragma gs attribute(int Timestamp[E])

// In general:
#pragma gs attribute(<type> <name>[<count>])
```

Listing 5.2: The `attribute` pragma instructs the GS Engine read buffer data from a file and make it accessible to the GLSL code.

```
// Allocate a buffer of floats for generated node data
#pragma gs scratch(float EarliestTimestamp[N])

// Allocate a buffer of ints for generated edge data
#pragma gs scratch(int AccountDifference[E])

// Allocate an atomic integer to count occurrences
#pragma gs scratch(atomic_uint Count)
```

Listing 5.3: The `scratch` directive allocates zero-initialized SSBO buffers for read-write access to the GLSL shaders.

```
// GS Program code
#pragma gs attribute(float X[1024])
#pragma gs shader(positional)
void main() {
    gl_Position.x = X[gs_NodeIndex];
}
```

Listing 5.4: Before transpilation (Listing 5.5), the GS Program has pragma statements to help generate standard GLSL code and parameters to control the GS Engine.

```
// GLSL Code
layout (std430, binding=3) buffer _GS_X_BUFFER {
    float X[1024];
};
void main() {
    gl_Position.x = X[gs_NodeIndex];
}

// GS Engine Code
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, buffer);

GLvoid *data = malloc(...);
glBufferData(GL_SHADER_STORAGE_BUFFER,
    1024 * sizeof(GLfloat), data, GL_STATIC_DRAW);

GLuint index = 3; // from GLSL layout binding
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, index, buffer);
```

Listing 5.5: After transpilation (Listing 5.4), the pragma statements are removed and the OpenGL code is instructed how to bind data buffers to the GLSL code.

```
#pragma gs attribute(float XPos[N])
#pragma gs attribute(float YPos[N])
#pragma gs shader(positional)
void main() {
    float x = XPos[gs_NodeIndex];
    float y = YPos[gs_NodeIndex];
    gs_NodePosition = vec2(x, y);
}
```

Listing 5.6: The GS Positional Shader for Figure 5.3.

Built-In Variables. In a positional shader, the only indices that can be directly operated on are those corresponding to graph nodes. The `gs_NodeIndex` variable can be used to look up attributes for the current node. The only required variable for this stage is the `gs_NodePosition` variable which specifies and controls the graph layout. Listing 5.7 shows the equivalent GLSL variable declarations for all built-in variables for the positional stage.

Relational Shader

Example. The GS relational shader operates between the positional and appearance shaders. The relational shader's `main` function is called once for each edge making the relational shader the ideal place to process edge attributes. It could also serve a dual purpose of processing node attributes based on the edge relations between them.

Listing 5.8 shows the relational shader used by the example in Figure 5.3 (Right). This relational shader compute the in- and out-degree of each node within the graph in parallel. This shader allows Figure 5.3 (Right) to test a relational hypothesis that packages may be "at risk" based on comparing the in- and out-degrees of JavaScript packages and dependencies.

For example, assuming that an edge indicates that the *source* node depends on the *destination* node, it is illustrative to consider the case when the source node is the React library, a common foundational package for JavaScript. As expected, React has a high in-degree. If React has a destination node (i.e. a dependency) that has virtually no direct dependencies (hence a low in-degree), React may be at higher risk because that dependency cannot be relied upon. Obviously, there can be many other formulation or variations of such hypotheses and that these hypotheses can be deeply nuanced and domain specific.

Usage. The GS Relational Shader is the ideal place to process every graph edge to compute new edge attributes or else to compute a reduction across all edge relations. Specifically examples include: (i) compute a new edge attribute for each edge individually; (ii) compute a new node attribute with atomic operations of each source and target node of an edge; (iii) compute a scalar reduction across all edges with atomic operations.

Built-In Variables. The input variables of importance to the relational shader include an index for the current edge, and two indices for the nodes that make up the current edge. The only predefined output variables are optional replacement positions for each node. If

omitted, the nodes will retain the positions calculated from the positional shader. Listing 5.9 lists all built-in variables for the relational stage.

Appearance Shader

Example. The appearance shader is responsible for computing the per-pixel color of the rendered image. As the final shader to be executed, no new attributes should be calculated. However, it is the correct place to perform the final filtering and compute the final summary statistics.

Usage. The GS Appearance Shader is the last stage and is therefore the ideal place to process the fully calculated node and edge attributes from earlier stages. An appearance shader will usually do one or more of: (i) compute the color of an edge based on node and edge attributes; (ii) filter edges based on node and edge attributes; and (iii) calculate total counts of nodes and edges with certain properties.

Built-In Variables. The input variables to this stage are the same as in the relational stage because every pixel corresponds to a single edge. The newly created node and edge attributes from the earlier stages are now settled and fully computed as well. Listing 5.11 lists all built-in variables for the appearance stage.

5.3 Graph Shaders: Applications & Use-Cases

In this section, I demonstrate three applications for the visualization and analysis of large graph data. These applications serve as a study of the effectiveness of the Graph Shader approach as a means to evaluate hypotheses about the nature of the graph data. Each application is intended to be illustrative of the kinds of evaluation possible, but is in no way exhaustive.

5.3.1 Overview

Graph Datasets. Table 5.1 lists the datasets used to evaluate GS, ranging between 1.2 to 3.8 million vertices and up to 63.5 million edges. The three datasets are used to demonstrate

```
#pragma gs shader(positional)
in int gs_NodeIndex;

out vec4 gs_NodePosition; // required
```

Listing 5.7: Built-in variables for the GS Positional Shader.

```
#pragma gs scratch(atomic_uint MaxOutDegree)
#pragma gs scratch(uint OutDegree[N])
#pragma gs scratch(atomic_uint MaxInDegree)
#pragma gs scratch(uint InDegree[N])
#pragma gs shader(relational)
void main() {
    uint od = 1 + atomicAdd(OutDegree[gs_SourceIndex], 1);
    atomicCounterMax(MaxOutDegree, od);

    uint id = 1 + atomicAdd(InDegree[gs_TargetIndex], 1);
    atomicCounterMax(MaxInDegree, id);
}
```

Listing 5.8: The GS Relational Shader for Figure 5.3 (B).

```
#pragma gs shader(relational)
in int gs_EdgeIndex;
in int gs_SourceIndex;
in int gs_TargetIndex;

in out vec4 gs_SourcePosition; // optional
in out vec4 gs_TargetPosition; // optional
```

Listing 5.9: Built-in variables for the GS Relational Shader.

```
#pragma gs attribute(uint Devs[N])
#pragma gs attribute(uint Vuln[N])
#pragma gs scratch(uint Seen[E])
#pragma gs scratch(atomic_uint TotalSeen)
#pragma gs scratch(atomic_uint TotalRisky)
#pragma gs scratch(atomic_uint TotalVuln)
#pragma gs scratch(atomic_uint TotalBoth)
#pragma gs shader(appearance)
void main() {
    uint dependents =
       InDegree[gs_SourceIndex] + InDegree[gs_TargetIndex];
    uint devs_can_support = 100 * Devs[gs_TargetIndex];
    bool risky = dependents > devs_can_support;
    bool vuln =
        (Vuln[gs_SourceIndex] + Vuln[gs_TargetIndex]) > 0;

    uint seen = atomicCounterAdd(Seen[gs_EdgeIndex], 1);
    if (seen == 0) {
        atomicCounterAdd(TotalSeen, 1);
        atomicCounterAdd(TotalRisky, uint(risky));
        atomicCounterAdd(TotalVuln, uint(vuln));
        atomicCounterAdd(TotalBoth, uint(risky || vuln));
    }

    gs_FragColor = vec4(0.1);
    gs_FragColor.r = float(vuln);
    gs_FragColor.g = float(risky);
}
```

Listing 5.10: The GS Appearance Shader for Figure 5.3 (Right), which highlights the maximum Out/In degree, based on calculations from the relational shader in Listing 5.8.

```
#pragma gs shader(appearance)
in uint gs_EdgeIndex;
in uint gs_SourceIndex;
in uint gs_TargetIndex;

out vec4 gs_FragColor; // required
```

Listing 5.11: Built-in variables for the GS Appearance Shader.

the rendering results in this section. The SO-Answers dataset is the largest and is used to demonstrate rendering performance as described in Section 5.4.

Domain specifics for each dataset are described in subsequent sections. Briefly, *JS-Deps* is a public dataset of inter-dependency relationships among all of the open-source JavaScript packages maintained by NPM [npm]; *SO-Answers* is a Stanford SNAP dataset of Question and Answer relationships from Stack Overflow [67]; *NBER-Patents* is a dataset of cross-citation relationships among US patents [44]. All three datasets contain domain-salient attributes, complex relationships, and an underlying temporal dimension.

Graph Layouts. There are many graph layout algorithms one can consider. In this work, extensive experiments were conducted for a variety of layout algorithms. For each algorithm, the layout was iteratively improved over the course of hours and days. The layout from the last iteration was saved and used for the visuals in this section.

Many of the well known algorithms did not perform well on the test datasets. Among them were *sfdp* and *OpenOrd (formerly: DRL)*, both are force based layout approaches and reported great layout results for graphs with hundreds thousands nodes. However, both *sfdp*[52] and *OpenOrd*[69] produced layouts that were visually unintelligible. These two methods also consumed too much time per iteration. In addition, *ARF* [41] and *GraphOpt* [Schmuhl] were also tested. Unfortunately neither could finish a layout run on the test datasets.

Large Graph Layout (LGL)[8] and *ForceAtlas2*[57] yielded satisfactory layout. While neither converged on the three large test datasets, they were able to proceed through multiple iterations in a sub-hourly time frames. *ForceAtlas2* was capable of 1000 iterations in an hour while *LGL* could produce 1000 iterations in minutes. With *ForceAtlas2* handling larger datasets better, it was chosen as the method for the NBER-Patents and the SO-Answers dataset. The JS-Deps dataset uses a layout generated by *LGL*.

5.3.2 Application: Software Dependency & Vulnerability

Driving Question. *Do open-source JavaScript packages with more maintainers have fewer vulnerabilities*? Open source has provided building blocks, frameworks, and infrastructure used in virtually all software projects. Developers can easily find and reuse existing open

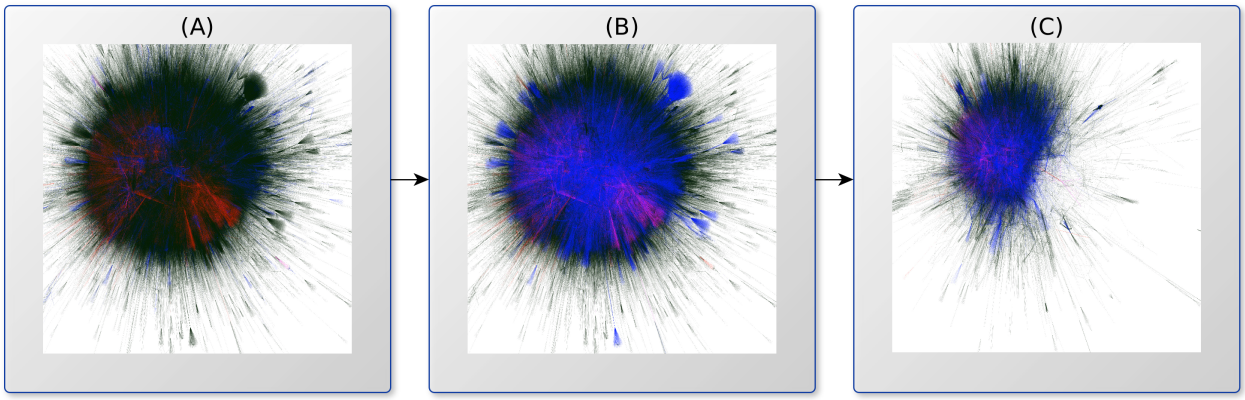

Figure 5.3: Using Graph Shaders (GS) to render an open-source project inter-dependency graph ($|N| = 1.2M$, $|E| = 6.2M$) extracted from NPM. In the examples, the GS shaders have under 70 lines of code with the goal of identifying and measuring the number of packages most at risk (technical details in Section 5.3.2). Like with OpenGL GLSL shaders, it is easy to experiment with GS shaders. Designing the GS shaders of these examples took about 5 minutes of iterative experiment. The typical render loop takes around 1 second at 2048×2048 pixel resolution on a GPU. Each example builds on the next with the goal of finding a property that correlates with whether a package is vulnerable. (A) maps existing graph attributes from the raw data to color channels: red if the package has a recorded vulnerability, blue if a package is "risky" and has a dependency with fewer developers than itself. (B) uses a GS Relational Shader to compute the in-degree graph property for each package to more accurately identify risky packages. (C) uses a GS Appearance Shader to focus on packages released during the year 2014.

source components and often provide the resulting software as open source too. Given the impact of the open source movement, the community trends within are of great importance, especially due to the network effect of vulnerabilities in one package affecting potentially thousands of downstream packages and potentially hundreds of thousands of end-users.

Available Attributes. The "JS-Deps" dataset is an early-2020 snapshot of the entire Node Package Manager (NPM) dependency graph. The $|N| = 1,169,747$ nodes represent JavaScript packages in the NPM registry and includes the date of last release (as a Unix timestamp), the number of maintainers of the package, and whether a vulnerability has been reported for the package. The $|E| = 6,188,869$ edges (notationally, (u, v)) represent dependencies between a package u and its dependency v. As an additional preprocessing step, each node has an X and Y position, generated from LGL (Section 5.3.1).

Graph Shader Design. The GS Program for the JS-Deps visualization in Figure 5.3 has already been explored in Section 5.2.2. To arrive at that visualization, I trialed many experiments with the raw graph data and with the iteratively designed visualizations. The overall goal of this visualization is to find a metric that is correlated with packages that are vulnerable, so that this new metric could help predict how "risky" a package is as a dependency in one's own project.

The first attempt at measuring risk (Figure 5.3 (A)) is to simply compare a package's number of developers with a dependency package's number of developers. Using atomic counter variables, the vulnerable percentage of the population of risky packages was found to be the same as the percentage for the entire population of packages (11%).

The second attempt at measuring risk (Figure 5.3 (B)) is to compute new graph attributes using a GS Relational Shader. For this data, it was chosen to calculate the in- and out-degree of each package. Working off of the hypothesis that a developer can only reasonably support a certain number of other developers with their software package, the risk measure in Listing 5.10 was chosen. This decision is supported numerically by comparing the vulnerable percentage of populations and finding that this new metric correlates with package vulnerabilities at a rate of 14% vs 11%.

The next iteration of the design process was to verify whether this property holds for smaller time spans (Figure 5.3 (C)). Using the GS Appearance Shader, packages that were

released in any year than 2014 were discarded. The vulnerable percentage was calculated again, though for a smaller population, and the result is a 15% rate of vulnerabilities in "risky" packages compared to the entire population's 11% rate. This iterative process serves to illustrate the experimental approach to designing Graph Shaders. Although the field of software ecosystems and vulnerability analysis is very complex, the use of Graph Shaders meant that several hypotheses could be tested and validated rapidly.

5.3.3 Application: Stack Overflow Q&A Networks

Driving Question. *On Q&A forums, do most questions get answered by veteran or new accounts*? Question and Answer website forums contain a complex ecosystem of user interactions as time passes. To answer this question, this section operates on a dataset of over 2 million Stack Overflow users who have asked a question, answered a question, or commented on a question or answer of another user.

Dataset Attributes. The "SO-Answers" dataset is a compilation of anonymized Stack Overflow user contributions over a period of 7.5 years, until March 6, 2016. Each individual answer-on-a-question, comment-on-a-question, and comment-on-an-answer is a "contribution." The $|N| = 2,601,977$ nodes represent individual user accounts and includes earliest & latest contribution (as a Unix timestamp) and in-degree & out-degree (contributions received and contributions made, respectively). The $|E| = 63,497,050$ edges (notationally, (u, v)) represent a contribution from account u to account v and includes when the contribution was made (as a Unix timestamp) and the "lifespan" of the account (the duration of time between first and last contribution). The layout uses FastAtlas2.

Graph Shader Design. To answer the driving question, it is necessary to determine what a "veteran" user is. Given the attributes within the dataset, two metrics were found experimentally: 1) whether account u makes many more contributions than they receive and 2) whether account u has a larger lifespan than account v. Edges are colored by how many metrics they satisfy: black if 0, red if 1, and yellow if 2. The graph is filtered by the date for u's first contribution.

The generated graph layout for the SO-Answers dataset proved to be incomprehensible using only visual qualities. Early in the design process, I found a need for an improved layout

```
// GS Program for SO-Answers dataset
#pragma gs attribute(float XPos[N]) // range [0, 1]
#pragma gs attribute(float YPos[N]) // range [0, 1]
#pragma gs attribute(uint Date[E]) // unix timestamp

#pragma gs scratch(uint MinDate[N]) // earliest post
#pragma gs scratch(uint MaxDate[N]) // most recent post
#pragma gs scratch(uint InDegree[N]) // dependents
#pragma gs scratch(uint OutDegree[N]) // dependencies

#pragma gs scratch(atomic_uint MaxInDegree)
#pragma gs scratch(atomic_uint MaxOutDegree)

#pragma gs shader(positional)
void main() {
    float x = XPos[gs_NodeIndex];
    float y = YPos[gs_NodeIndex];
    gs_NodePosition = vec2(x, y);
}

#pragma gs shader(relational)
void main() {
    // same InDegree/OutDegree code as in Listing 5.8

    uint date = Date[gs_EdgeIndex];

    atomicCompSwap(MinDate[gs_SourceIndex], 0, date);
    atomicCompSwap(MinDate[gs_TargetIndex], 0, date);

    atomicMin(MinDate[gs_SourceIndex], date);
    atomicMin(MinDate[gs_TargetIndex], date);

    atomicMax(MaxDate[gs_SourceIndex], date);
    atomicMax(MaxDate[gs_TargetIndex], date);
}

#pragma gs shader(appearance)
void main() {
    uint od = OutDegree[gs_SourceIndex];
    uint id = InDegree[gs_SourceIndex];
    bool posts_a_lot = od >= id / 32;

    uint my_span =
        MaxDate[gs_SourceIndex] - MinDate[gs_SourceIndex];
    uint your_span =
        MaxDate[gs_TargetIndex] - MinDate[gs_TargetIndex];
    bool less_veteran = my_span < your_span;

    gs_FragColor = vec4(0.1);
    gs_FragColor.r = float(posts_a_lot || less_veteran);
    gs_FragColor.g = float(posts_a_lot && less_veteran);
}
```

Listing 5.12: The GS Program for the SO-Answers shader (Figure 5.4).

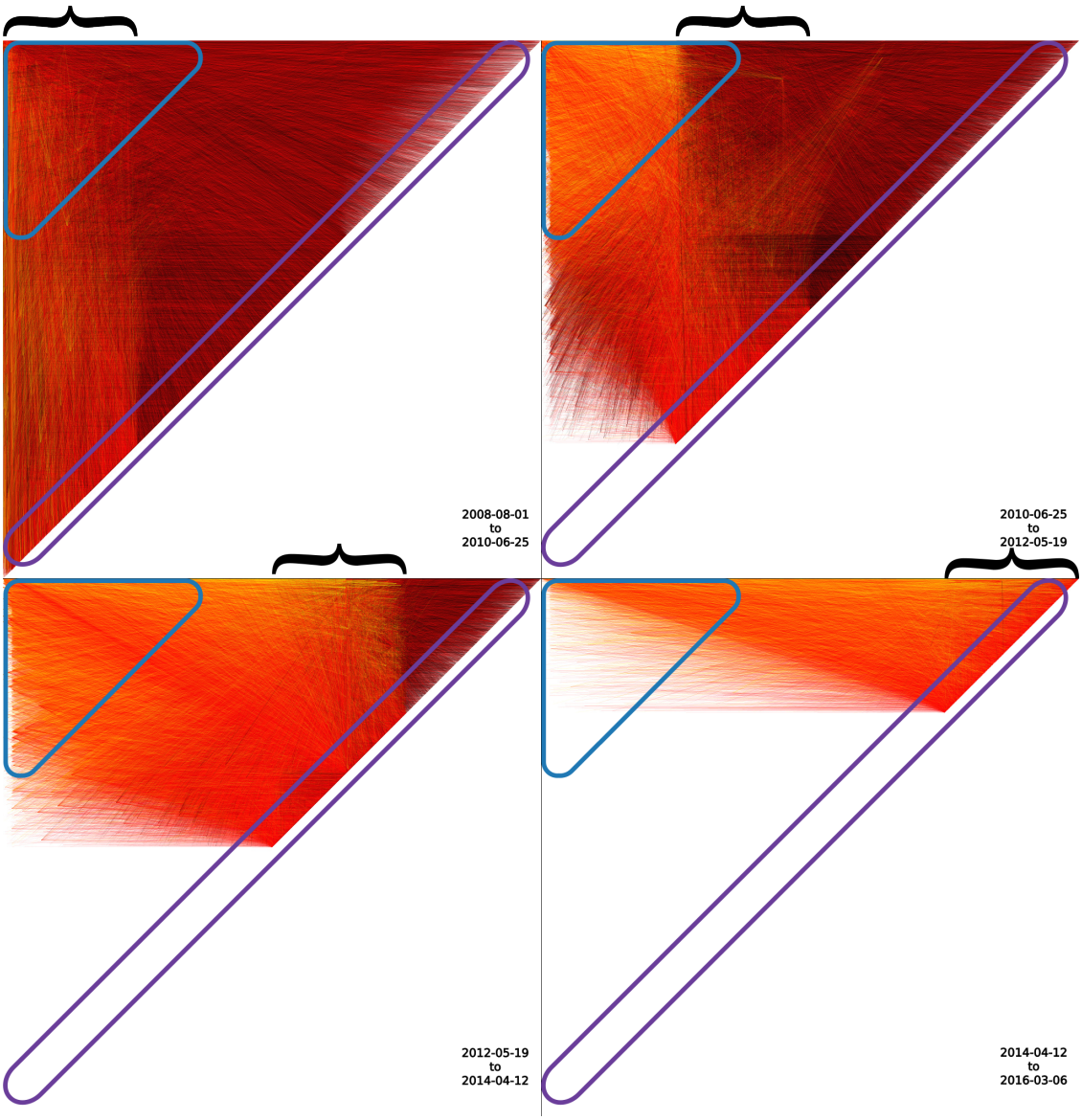

Figure 5.4: SO-Answers (with "detangled" layout): The new layout positions nodes on when their first and last contributions occurred. By design, accounts (nodes) within the corner (blue triangle) are more veteran than accounts along the edge (purple rectangle).

that conveys multiple properties of the graph at once. This need prompted a generated layout that draws inspiration from span-space visualization [101], by positioning users according to when their first (x-coordinate) and last (y-coordinate) contributions occurred.

Figure 5.4 shows the improved layout for the SO-Answers dataset. In the figure, each quadrant represents the edges whose source node's x-coordinate is within a range represented by a brace symbol. Additionally, characteristics important to the driving question can now be mapped to location within the chart: older accounts are in the corner (blue triangle) and accounts with a smaller lifespan are along the diagonal (purple rectangle).

5.3.4 Application: Patent Citation Networks

Driving Question. *How does the lag time between patent application and patent acceptance change over time and between different categories*? Patents and the evolution of patents provide a unique history of innovation. Specifically, that's captured in the patent citation network, which spans across all industry fields.

Dataset Attributes. The "NBER-Patents" dataset of patents and their citation network comes from the National Bureau of Economic Research (NBER) Patent Data Project. The dataset is publicly available and includes both the patents and the citations of patents over the time period from 1975 to 2006. The $|N| = 3,209,376$ nodes represent individual patents and includes when the year of the application, the year that the patent was granted, the coarse category ($\# = 6$), and the granular subcategory ($\# = 800$). The $|E| = 23,650,891$ edges (notationally, (u, v)) represent citations from patent u to existing patent v.

Graph Shader Design. Categorical data is an area where the efficient design of GS is especially beneficial. By design, the driving question can be simplified into individual shaders and renderings that address a single aspect of the question. In Figure 5.5, each subplot is the result of a parameterized Graph Shader that answers a smaller question. For example, the subplot in column 2 and row 4 answers the question *what is the "patent lag time" from 1975 to 1979 for the set of computers & communication patents that cite electrical patents*? To map patent lag time to visual appearance, the shader uses a simplistic color map where

```
// GS Program for NBER-Patents dataset
#pragma gs attribute(float XPos[N])
#pragma gs attribute(float YPos[N])
#pragma gs attribute(uint AppYear[N])
#pragma gs attribute(uint GotYear[N])
#pragma gs attribute(uint Cat1[N])
#pragma gs attribute(uint Cat2[N])

#pragma gs shader(positional)
void main() {
    float x = XPos[gs_NodeIndex];
    float y = YPos[gs_NodeIndex];
    gs_NodePosition = vec2(x, y);
}

#pragma gs shader(relational)
void main() { /* no relational shader used */ }

#pragma gs shader(appearance)
#pragma gs scratch(uint Seen[N])
#pragma gs scratch(atomic_uint count_bad_source_cat)
#pragma gs scratch(atomic_uint count_bad_target_cat)
#pragma gs scratch(atomic_uint count_too_early)
#pragma gs scratch(atomic_uint count_too_late)
void main() {
    bool bad_source_category = Cat1[gs_SourceIndex] != 2;
    bool bad_target_category = Cat1[gs_TargetIndex] != 4;
    bool too_early = AppYear[gs_SourceIndex] < 1985;
    bool too_late = AppYear[gs_SourceIndex] > 1990;

    uint seen = atomicCounterAdd(Seen[gs_EdgeIndex], 1);
    if (seen == 0) {
        atomicCounterAdd(count_bad_source_category,
            uint(bad_source_category));
        atomicCounterAdd(count_bad_target_category,
            uint(bad_target_category));
        atomicCounterAdd(count_too_early, uint(too_early));
        atomicCounterAdd(count_too_late, uint(too_late));
    }

    if (bad_source_category || bad_target_category
        || too_early || too_late)
        discard;

    int lag = GotYear[gs_SourceIndex] - AppYear[gs_SourceIndex];

    gs_FragColor = vec4(0.1);
    gs_FragColor.r = float(lag < 2);
    gs_FragColor.g = float(2 <= lag && lag < 4);
    gs_FragColor.b = float(4 <= lag);
}
```

Listing 5.13: The GS Program for the NBER-Patents dataset (Figure 5.5).

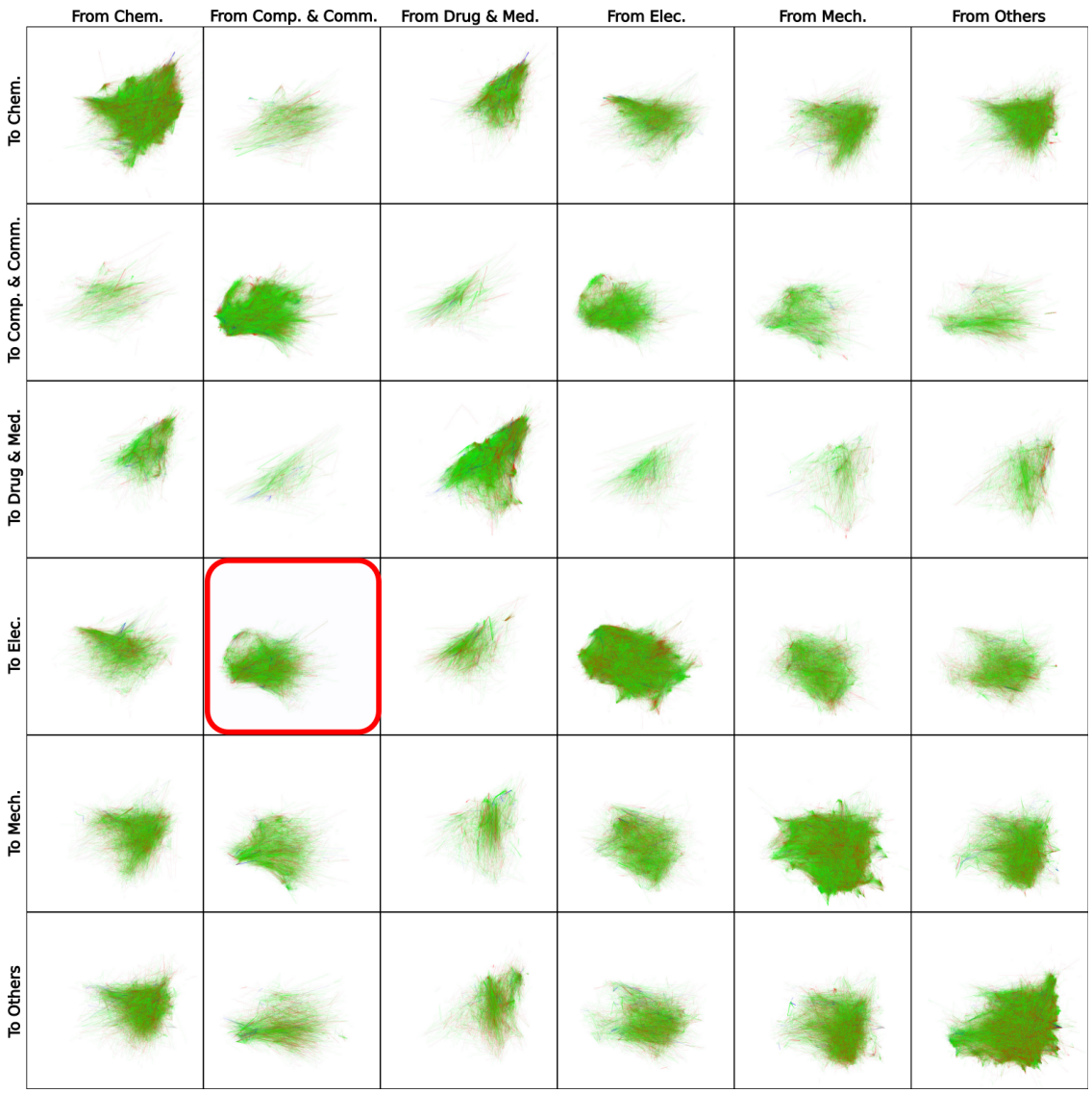

Figure 5.5: NBER-Patents: Citations (edges) between patents (nodes) are colored: red if lag time is 0 or 1 years; green if lag time is 2 or 3 years; and blue if lag time is 4 or more years. The included shader is for the highlighted subplot, representing citations from "computers and communication" patents to "electrical" patents whose application year was between 1975 and 1979.

a lag time between 0 and 1 years is red, between 2 and 3 years is green, and 4 or more years is blue.

5.4 Graph Shaders: Performance Study

To demonstrate the performance of the entire Graph Shaders system, several performance studies were conducted. Each study aims to explore the performance of some component of GS.

The **GS Engine** is the lowest level component of GS. In Section 5.4.1, I study the performance of this component to provide a baseline level of performance for the other studies in this section.

The **GS Service** is the next component that can be studied. In Section 5.4.2, I study the performance of the service in a stress-testing and overloaded scenario. This is intended to demonstrate the maximum performance achievable.

The **GS User** is a more theoretical component and is mainly concerned with the performance under normal conditions of user interaction. In Section 5.4.3, the intent is to provide proof that the GS Service is capable of supporting an interactive user experience when under load from multiple users simultaneously.

In each of the following sections, the testing environment is made up of two parts. The server-side environment is used in all tests and is largely affected by configuration for the GS Service. In Section 5.4.3, the performance study involves the client-side and network environments.

Server-Side Test Environment. The GS Service runs on a rack-mounted server with a NVIDIA® M6000 GPU with 24GB of on-board memory, dual Intel Xeon (E5-2650 v4, 12-core, 2.2 GHz) CPUs, and 128GB of RAM. Service deployment is through Docker. It is expected that rendering performance is GPU bound and not CPU bound. As a result, the most important facet of the server-side environment is the GPU.

GS Service Configuration. Each study uses the same configuration to make comparison between studies possible. This configuration is: a single GS Service instance rendering tiles of size 256×256 pixels, for an overall composite resolution of 1920×1200

pixels. This results in between $8 \times 5 = 40$ and $9 \times 6 = 54$ tiles to make up the final image. The rendering performance tests are run using the SO-Answers dataset, which is the largest of the datasets prepared for this work. SO-Answers is a graph with 2.6 million vertices and 63 million edges.

Client-Side Test Environment. For interacting with the GS Service, the client-side interface is run from a modest HP Elitebook 840 G5 notebook computer with an Intel® Core™ i5-8350U CPU running at 1.70GHz and 32GB of RAM using a recent version of the Chrome web browser (version 99.0.4844.82). The choice of notebook reflects a commonly used remote work environment.

Network Test Environment. The server-side connection to the Internet is a 1000Mbps network. The client-side connection to the internet is a residential 600Mbps Wi-Fi network. Round-trip latency between client-side and server-side environments measured by the Linux ping command are min/avg/max = 58/61/68ms.

5.4.1 GS Rendering Engine Study

After GS shaders are transpiled into OpenGL GLSL shaders, the rendering process is the same as any other OpenGL code that uses GLSL shaders. From that respect, there are no noticeable performance differences. Because the transpiling process deals with only source code, usually less than 100 lines, the transpiling process takes a negligible amount of time as well.

For this test, the GS Engine is implemented as a headless renderer, which co-exists on the same rack-mount server that hosts the graph datasets. That server runs Linux and has a NVIDIA® M6000 GPU with 24GB of on-board memory, dual Intel Xeon (E5-2650 v4, 12-core, 2.2 GHz) CPUs, and 128GB of RAM. At 2048×2048 image resolution, the pure rendering time (excluding I/O time) ranges between 0.5 and 1.5 seconds. Specific rendering times are shown in Table 5.1.

Table 5.1: Timing Results (2048 × 2048 image resolution)

Name	**Nodes**	**Edges**	**Seconds/Frame**
JS-Deps	1.2M	6.2M	1.1 - 1.3
SO-Answers	2.6M	63.5M	2.6 - 2.9
NBER-Patents	3.8M	16.5M	2.1 - 2.9

5.4.2 GS Rendering Service Study

The study in this section is designed to understand the GS Service's performance when under heavy load. This stress test makes many simultaneous requests to the service at the same time and measures the time needed for the service to process all requests.

Four specific scenarios are tested that vary A) the number of simultaneous requests and B) the access patterns of those requests.

Simultaneous Requests. When using the GS Service from outside of a browser, as in this test, the number of active requests at one time can be varied. *Scenarios 1A and 1B* enable 6 concurrent requests at the same time, simulating the load of a single user trying to render a single high-resolution image. The number 6 is based on the common browser limit for concurrent requests. *Scenarios 2A and 2B* enable 120 concurrent requests, simulating the load of 20 users each with 6 concurrent requests.

Access Patterns. There are two primary patterns when accessing the tiled graph rendering that the GS Service provides. *Scenarios 1A and 2A* use sequential access to render a single high-resolution of a graph in the common top-to-bottom, left-to-right order. The primary independent variable for this access is the total resolution of the final rendered image, and dependently, the number of tiles being requested. Second, *Scenarios 1B and 2B* use random access to render independently sampled tiles (at 256×256 resolution) with replacement. This simulates the access patterns of a service being used by many different users with different goals and motives.

Scenario 1 (A and B). This scenario tests the GS Service's performance when used as a component of a larger system. For example, a scientist wanting to see and share the full view of the graph as a static image could use GS to render that static image without having to directly interact with the web interface. In this case, interactivity is of less concern, and the overall throughput of requests per second is more important.

The server-side scalability results for Scenario 1 are in Figure 5.6. Each test case was run at least 20 times to give more accurate results. As expected, the results find that higher concurrency (number of active requests) yields higher performance, especially with larger numbers of total requests (i.e. a larger final resolution). It was also found that having more

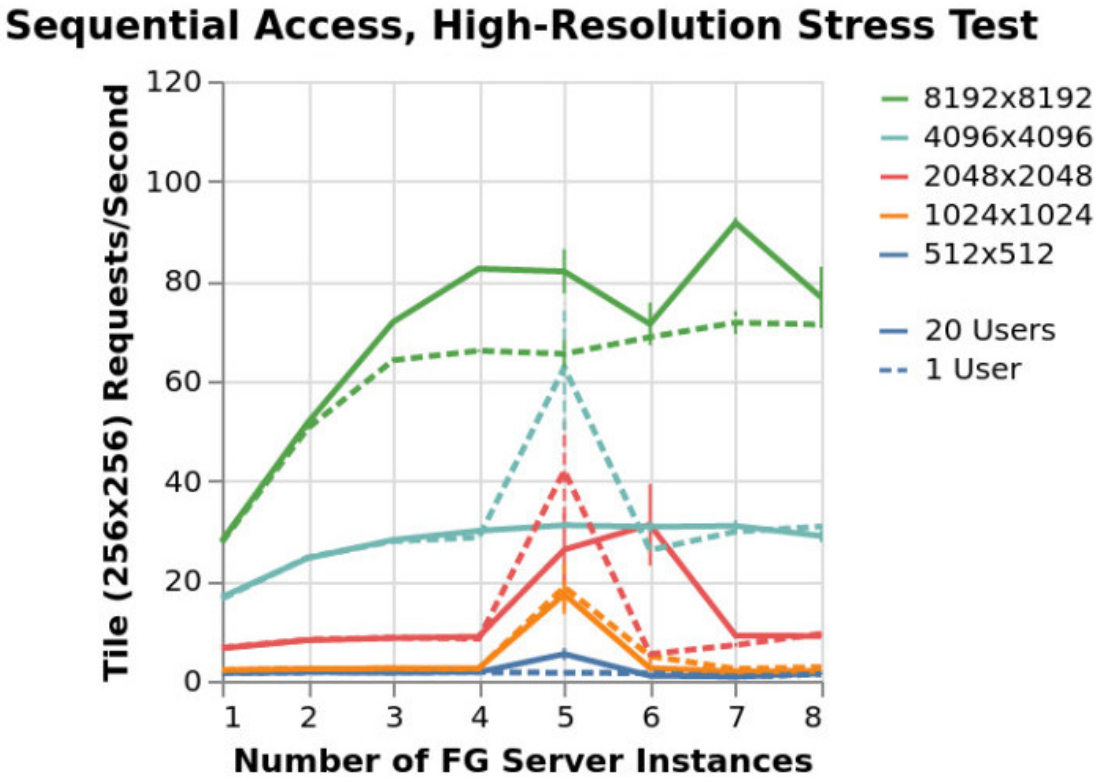

Figure 5.6: Server-side stress-test results for Scenario 1, comparing (A) 1 simulated user and (B) 20 simulated users. Scenario 1 involves rendering a single high-resolution image of the whole graph by stitching together multiple tiles ($256x256$). For example, the $8192x8192$ image requires 1024 separate tiles, each tile requires a separate rendering request.

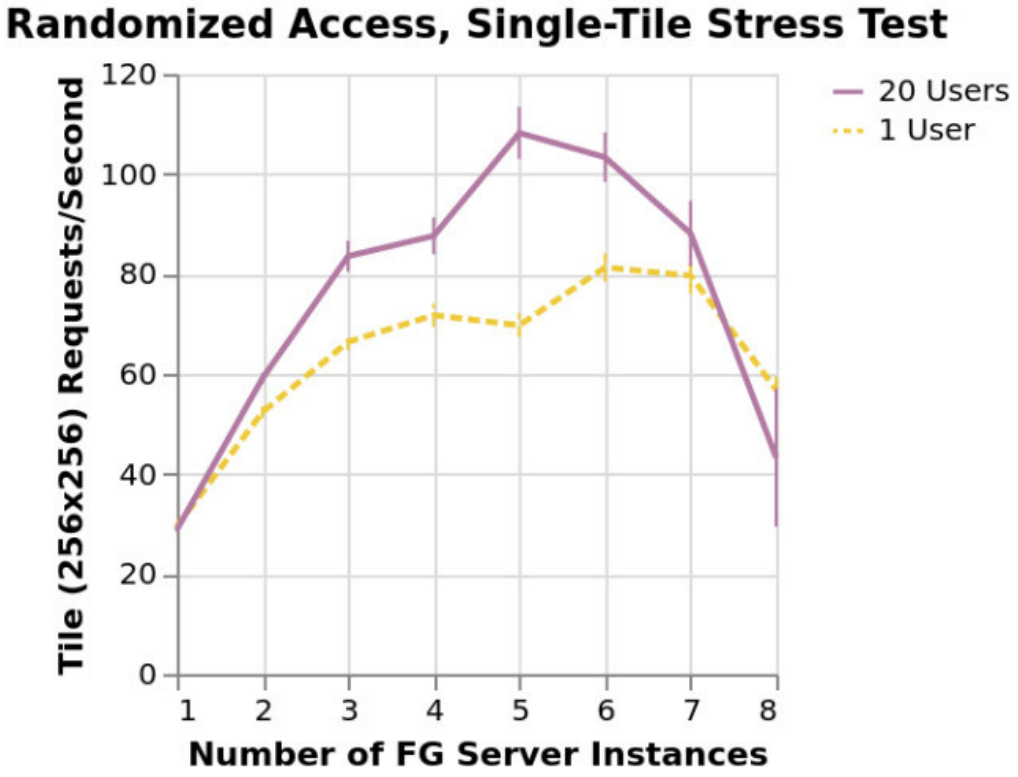

Figure 5.7: Server-side stress-test results for Scenario 2, comparing (A) 1 simulated user and (B) 20 simulated users. Scenario 2 tests independent requests for a randomized image tile. The results indicate GS is capable of rendering up to 110 tiles per second and this optimum is achieved at 5 FG instances on one M6000 graphics card. Using more than 5 instances yields worse results as we reach an area of diminishing returns.

instances running at a time gives better performance, although past 5 instances, there are diminishing returns.

Scenario 2 (A and B). This scenario differs from Scenario 1 in that the tile requests are made randomly across many different zoom levels and X & Y position. By design, some requests are made for lower zoom levels (requiring many data partitions to render a single tile) and some are made for higher zoom levels (often using only a single data partition). This is in contrast with Scenario 1 where every area of the graph will be rendered by the end of the test. This characteristic tends to give higher request per second throughput than the equivalent Scenario 1 test.

The server-side scalability results for Scenario 2 are in Figure 5.7. As before, each test case is run 20 times to give a more accurate view of the average throughput. These tests were run for 1 simulated user and for 20 simulated users. The results indicate a higher average throughput than in Scenario 1, as expected, and the same diminishing returns past 5 instances.

5.4.3 GS Simulated-User Study

The study in this section is intended to understand the performance as seen by a simulated user. Unlike in the previous studies, the access patterns in this test are replayed from existing user interactions enabling a repeatable and automated method of testing. These interactions were chosen ahead-of-time and include changes to the shaders, zooming into and out of the graph, and panning around the graph. The total number of interactions in the test script is chosen so that the script takes around 1 minute to complete. In total, the scripted test is repeated 3 times on each of the 3 datasets.

The request-response latency of each individual tile is measured from the client-side perspective and combined with server-side measurements of rendering time and encoding time. The latter two measurements make up most of the server-side processing time and they demonstrate pipeline parallelism such that the CPU-limited image encoding of one response happens concurrently with the GPU-limited rendering of the next request.

Different interactions have different expected request-response latencies, so the results are grouped by interaction types. The tested interaction types include: *initial* full-screen rendering ($40 - 54$ tiles), *shader* changes ($40 - 54$ tiles), *zoom* in and out ($40 - 54$ tiles), *panning* translations ($20 - 27$ tiles).

Each of these interactions made on the client-side triggers a repeatable series of tile requests to the server and responses, in the form of rendered tiles, from the server-side. The requests and responses are then measured from both the client- and server-side perspectives, accounting for the end-user experience (including latency between client and server) and best-case performance (with zero latency).

The client-side perspective of performance results of the client-side test are in Figure 5.8. As expected, interactions that initiate more tile requests (*initial*, *shader*, *zoom*) have higher maximum latency. However, the time for the client to receive the very first tile rendered in response to a request is also an important characteristic. That metric is Time-to-First-Response (TTFR), which is of course also the minimum latency of each interaction. In every test case shown in Figure 5.8, the TTFR is at or below one second.

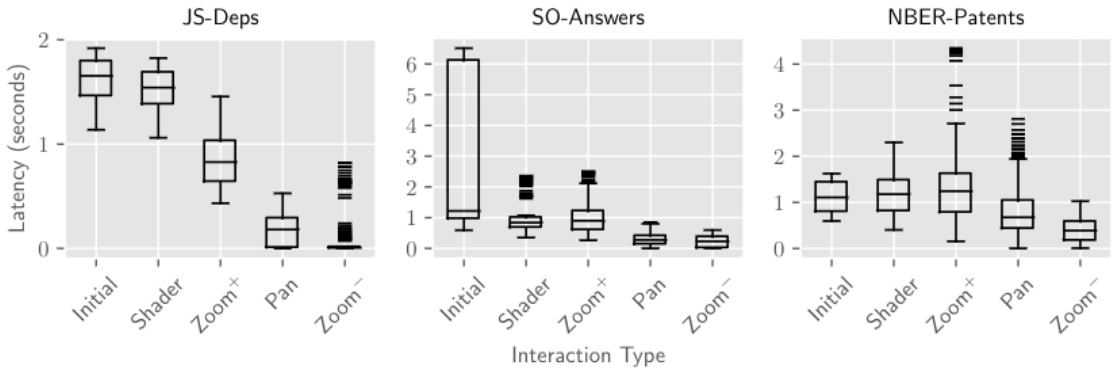

Figure 5.8: Client-side interactivity test results. Latency is measured as the time between the beginning of the rendering request and the end of the rendering response. Each request is made at the same time, although requests are processed in serial on the server, so these results are the total latency from the user's perspective.

Chapter 6

Integrative Scientific Applications using VaaS Services

6.1 Overview

In the preceding chapters, I have demonstrated applications and use-cases that adapt the VaaS architecture to new application domains including parallel flow (Chapter 3), mixed-reality (Chapter 4), and large graph (Chapter 5) visualization. Each application domain, respectively, highlights the need for a general solution to data-parallel, multi-device, and hypothesis-oriented VaaS.

In this chapter, I study the design and implementation of a VaaS that integrates multiple scientific VaaS together into a common and flexible framework. This new VaaS is called Weaver which acts as my final study of expressive and versatile VaaS using the capabilities I have proposed.

The purpose of Weaver is to be analogous to the metaphor of weaving services, requests, and responses into one cohesive system. In this way, Weaver is not intended to be a re-implementation of each VaaS into a single monolithic application, but instead to be another VaaS that is more expressive and versatile than the parts that make it up.

To demonstrate Weaver, I will study a representative application that is one of many possible applications that could be made with Weaver. The specific application I create is contrived, but is based on existing ideas within the field of climate modeling.

The specific application in this chapter is designed to explore the relationships within a flow visualization of a climate field. To model the flows and identify patterns, I have chosen to simply quantize the flow positions into a limited number of voxels, each representing a range of latitude, longitude, and pressure values. These voxels are then used to construct a graph, representing each voxel as a node, and each edge as the number of traces that go from one voxel to another during the course of the user's interactions.

Although simplistic, this application exercises every part of Weaver and demonstrates how Weaver is capable of supporting data-parallel, multi-device, and hypothesis-oriented visualization needs.

A core principle behind Weaver is its intention as a tool that integrates into an interactive notebook environment. For this demonstration, I demonstrate the use of Weaver in a JavaScript notebook environment called Observable, that is reminiscent of Jupyter notebooks but instead of running Python on a server or laptop, it runs JavaScript in a web browser. By design, Observable is designed for and suited to lightweight InfoVis applications. However, using Weaver, complex scientific visualization is just as accessible and easy to use as the more common InfoVis.

The remainder of this chapter is organized in the following way. In Section 6.2, I describe the design and implementation of Weaver and how it relates to the other chapters of this dissertation. In Section 6.3, I describe the design of an application for a simple form of climate modeling. I conclude this chapter in Section 6.4 with a preliminary study on the usability and performance of Weaver.

6.2 Weaver: Design Considerations

Weaver is designed to be integrative of existing VaaS while still supporting a hypothesis-oriented approach to visualization needs. The integrative needs of Weaver pose several challenges in its design.

The integration of multiple VaaS requires a flexible method of defining how and what to integrate. In Weaver, this is considering as "interfacing" code and is intended to adapt between the user's requests and the service's interface. In many cases, this interfacing

code simply defines where the other VaaS is located and what kinds of inputs and outputs it produces. However, in more complex cases, the interfacing code can completely modify the request structure to allow flexibility for the user.

The flexibility of data transformation requires a method of specifying the flow of data between services with some degree of pre- and post-processing. Although the data flow and API calls could be defined within a simple and declarative JSON structure, the structure itself is an ad-hoc instance of a Domain Specific Language (DSL). Colloquially, this general idea is encapsulated in the aprocryphal Greenspun's Tenth Rule: a warning against creating a unique DSL for an application instead of using an existing and fully featured language. For Weaver, I have chosen to represent data transformation workflows within an integrative VaaS by embedding an existing embeddable DSL.

The runtime of user code needs to be constrained with resource limits so that a single user cannot exhaust the server's resources on a single request. Outside of intentionally malicious requests, it is also possible and common for programs to have bugs that result in infinite loops and never halt. Although this problem is unsolvable in general, Weaver can make the choice of an interpreted DSL with built-in resource limits available. Specifically, the memory usage, processor cycles, and system call capabilities must be constrained.

The state of running applications needs to coexist on the server so that it can be accessible to other requests and other server instances. Although a VaaS is most scalable and performant when it is stateless, the state for the user-facing interface must still exist and be managed somehow. In the past [90, 86], this stateful information was managed completely in the JavaScript code running in the browser. However, Weaver makes the stateful data management from Chapter 4 available. The result of this decision is the shift of stateful information from the frontend code to the backend infrastructure where it can be accessed from multiple devices at the same time.

The interface of integrative VaaS needs to be dynamic, flexible, and interactive to support a wide range of applications. I have chosen to embrace the web platform as Weaver's interface. There are two primary advantages to this design. First, there is an intrinsic selection of audience: Weaver is an application for developers and scientists to simplify the challenge of "weaving" different services together. Second, notebook environments naturally

lend themselves to experimentation and collaboration. While a standalone application may be seen as finite and limited to the functionality that the original developer intends, a notebook is designed to be understood and modified.

6.2.1 Weaver: Implementation

According to the design needs of Weaver, I have made several decisions in its implementation (overview in Figure 6.1). Although other implementations of the idea behind Weaver are possible, and perhaps even provide better performance, the implementation in this section is optimized for developers to easily integrate new VaaS or design new applications and use cases.

The Weaver service is implemented in Python using the `Flask` library for its HTTP interface. To make external calls to other VaaS, the service uses the standard `requests` module, transporting each call over an HTTP interface. Weaver has no need for static file hosting, meaning that the server instance can be hosted on any platform and server without need for special configuration. For these API-only use cases, Flask is an optimal choice, especially for scientific VaaS.

Weaver's primary interface is an HTTP endpoint called `/weave/` that accepts a program written in the Lua programming language. The choice of language is another design decision that can be provided by any language. This program is executed on the server using Lua's native resource limits system inside of a "sandbox" where the program is unable to affect other parts of the service or the machine. Within the Lua environment, developer-written interfacing code is exposed to the user-written code as native Lua functions. These functions are designed to transparently switch execution between Lua and Python.

It is important to note that Weaver is designed to be flexible and dynamic. Although I have made specific choices in its design, the resulting interface and data flow is not hard-coded within the VaaS itself, but is determined from the user's requests. This design is in contrast with existing VaaS applications with a restrictive interface between a single kind of input and a single kind of output. Examples include Tapestry's "render one scientific volume with one camera configuration" or Braid's "produce one flow trace from one set of seeds."

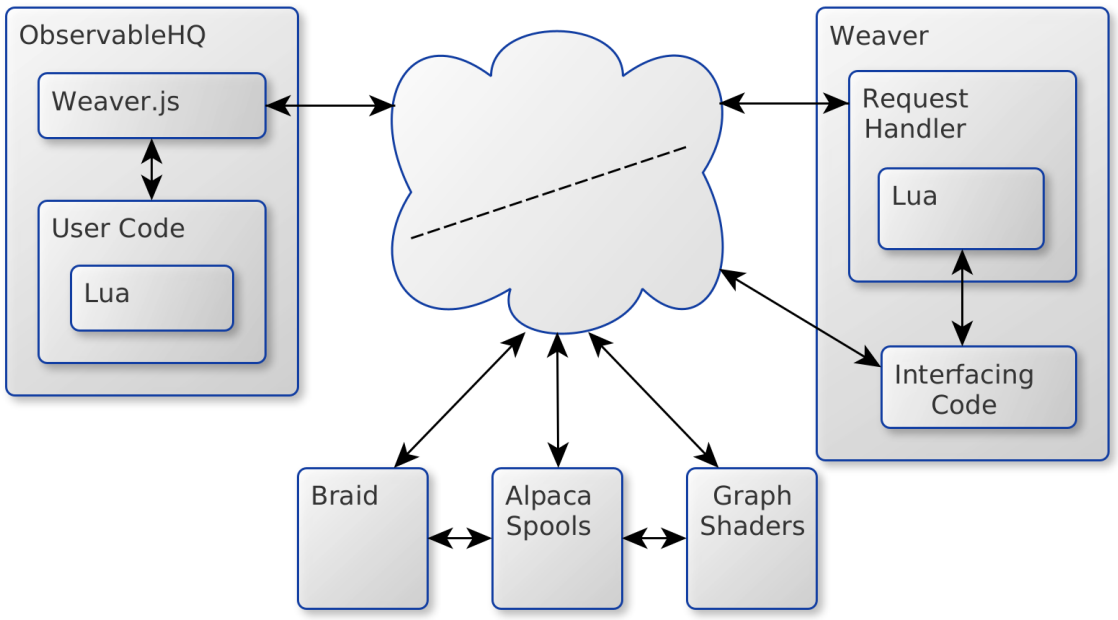

Figure 6.1: The Weaver system architecture presents a use case where the Weaver VaaS serves as the entrypoint to a public or private cloud of VaaS services. Lua code is embedded within the cells of an ObservableHQ JavaScript notebook. This code is processed by the Weaver server and executed with all of the interfacing functions as a part of Lua's global namespace, as ordinary Lua functions.

In contrast, Weaver has a much more flexible interface, namely "produce some output from some input." Every aspect of the input and output is configurable.

As a point of comparison, the VaaS services in the previous chapters can be implemented in the Weaver model with a single line of code. Using the Braid VaaS as an example, the original interface accepts the starting coordinates for flow visualization as a JSON object within an HTTP POST request and produces a JSON object of every subsequent flow coordinates. The Weaver interfacing code is exposed to the user-written code as a function that accepts coordinates as a Lua object and produces a Lua object with the flow coordinates. To the user, the exact implementation of this interface is inconsequential and can be transparently improved over time. Reductively, the original Braid VaaS can be re-implemented within Weaver with a single line of code (Listing 6.1).

This code could be customized or generated from a template for each user request. For example, using the JavaScript-native "tagged template literals," this specific request could be easily adapted according to the user's interactions. The advantage of tagged template literals is that they are an extensible mechanism for integrating multiple languages within the same notebook.

A complete sample of the notebook interface using template literals is in Figure 6.2. The notebook interface is made of multiple cells. The first cell creates an instance of the Weaver client-side library that manages the HTTP connections to the Weaver VaaS. The second cell defines an user interface with 3 sliders to select the starting latitude, longitude, and pressure values. The third cell debounces the user interface inputs so the other cells are only recalculated after the inputs are done changing. The last cell is responsible for using Weaver to trace the seed position. The client-side library exposes a `tag` function that pre-processes the backtick-delimited string that follows it. Within the template string, variables within `${}` are evaluated and replaced with their contents.

The example shown in Figure 6.2 illustrates the simplicity inherent in applications designed with Weaver. Although the depicted sample serves as a basic representation, more intricate applications adhere to the same underlying principles. These sophisticated applications, which will be further explored in the next section, can consist of tens of cells and hundreds of lines of code. Remarkably, the actual Weaver-interfacing code required to

```
return vaas_braid_trace{
  from="2017-01-01",
  to="2017-06-30",
  seeds={
    { lat=35.9606, lng=83.9207, prs=800.0 },
  },
}
```

Listing 6.1: Weaver can easily "re-implement" existing VaaS into its DSL-based request format. Within the code, variable values can be templated to reflect the current application state.

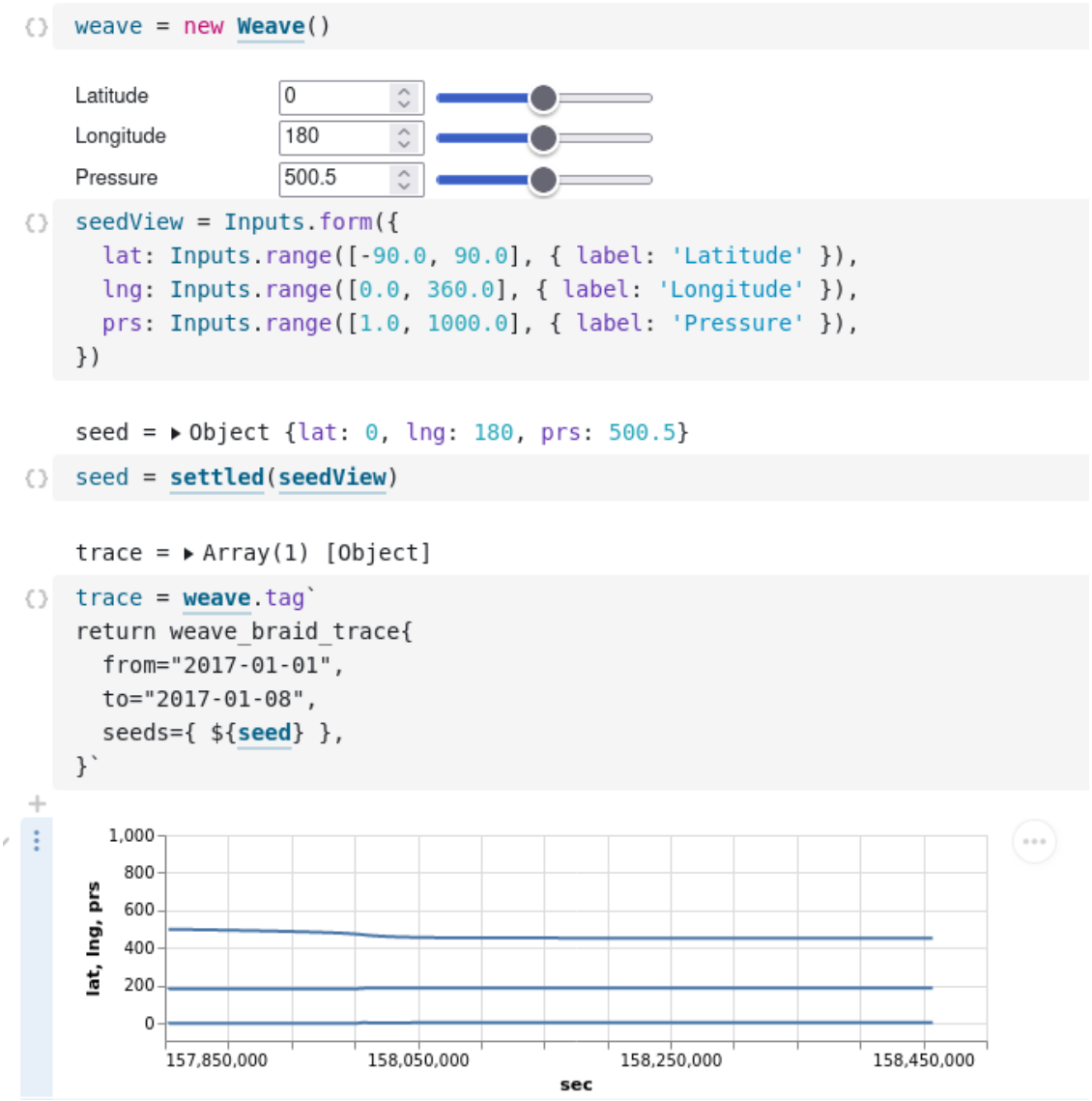

Figure 6.2: The Weaver frontend interface is a JavaScript notebook built in the ObservableHQ platform, similar in concept to Jupyter notebooks for Python developers. This screenshot includes the use of a client-side library for making API requests to the Weaver VaaS, the definition of a simple user interface, and a sample of Lua code used as in the request to Weaver.

achieve this functionality remains relatively concise, typically spanning only a few tens of lines.

6.3 Weaver: Applications & Use Cases

To demonstrate Weaver's effectiveness for integrative scientific visualization, I have created an illustrative, albeit contrived, application that makes use of all VaaS of the previous chapters. This section describes the specifics of an application called ClimateExplorer for interpreting climate flow visualization in a graph context. In essence, its goal is to be a simplistic way to model the transport of gasses in the atmosphere.

The only dataset that ClimateExplorer uses is a read-only copy of the NCAR dataset used in Chapter 3. This data represents a year-long observation of Earth's 3D atmospheric flow with $1,463$ discrete time steps, a spatial resolution of $720 \times 361 \times 36$, and a per-volume size of 150 GB. In total, there are 10 volumes, though the demonstrations in this section only use a single volume.

The frontend interface of ClimateExplorer is a JavaScript application built as a JavaScript notebook that features two modes of operation. The first mode is to perform interactive tracing across a 2D representation of the Earth. The second mode is a programmable, batch interface to flow tracing. In each case, the flow lines are used to immediately update the interface to show the newly traced lines. Transparently, the flow lines are post-processed on the server side to generate a graph dataset.

The ClimateExplorer application explores a novel and simplistic means of interpreting flow visualization as a graph. The core idea revolves around the inherent complex interpretation of flow visualization and a need to quantifiable measure the flow. The implementation of this "flow-to-graph" conversion is to quantize the flow into different voxels and add edge connections based on the flow path.

The specific functionality behind this interpretation of the flow as a graph is all user-defined in its own set of Lua functions. For example, to quantize a point of latitude (`lat`), longitude (`lng`), and pressure (`prs`) into its corresponding voxel the code in Listing 6.2 can be used and referenced within multiple requests to the Weaver VaaS.

```
function float2quant(options)
    local x = options[1]
    local lo = options.lo
    local hi = options.hi
    local bins = options.bins

    return math.floor((x - lo) / (hi - lo) * bins)
end

local PRS = { bins=10, lo=   1.0, hi= 1000.0 }
local LNG = { bins=10, lo=   0.0, hi=  360.0 }
local LAT = { bins=10, lo= -90.0, hi=   90.0 }

function point2voxel(options)
  local point = options[1]
  local prs = point.prs
  local lng = point.lng
  local lat = point.lat

  prs = float2quant{ prs, bins=PRS.bins, lo=PRS.lo, hi=PRS.hi }
  lng = float2quant{ lng, bins=LNG.bins, lo=LNG.lo, hi=LNG.hi }
  lat = float2quant{ lat, bins=LAT.bins, lo=LAT.lo, hi=LAT.hi }

  return PRS.bins * LNG.bins * prs + LNG.bins * lng + lat
end
```

Listing 6.2: To support a complex library of functions, Weaver requests include two codes: a library ("lib") and the current code to be run ("run"). This is a sample of "lib" code used within the Climate Modeling application for Weaver and is responsible for the exact interpretation of flow lines as graph data.

```
local seeds = {
  { lat=35.9606, lng=83.9207, prs=800.0 },
  { lat=36.9606, lng=83.9207, prs=800.0 },
  { lat=37.9606, lng=83.9207, prs=800.0 },
}

local traces = weave_braid_trace{
  from="2017-01-01",
  to="2017-06-30",
  seeds=seeds,
}

local edges = weave_spool_create{ "edges", {src="I"}, {dst="I"} }

for trace in traces do
  local previous = nil
  for point in trace do
    local voxel = point2voxel{ point }

    if previous != nil then
      vaas_spool_emit{ edges, src=previous, dst=voxel }
    end

    previous = voxel
  end
end

return edges
```

Listing 6.3: Within the Weaver climate modeling application, the main way that distributed state is generated is with this code which interfaces with the flow tracing code from Braid (Chapter 3). The resulting flows are post-processed on the server and re-interpreted as graph data. The exact functionality is defined in Listing 6.2.

In terms of visualization, ClimateExplorer is built to integrate parallel flow and graph visualization APIs. These two integrations are representative of a VaaS that primarily produces data and a VaaS that primarily consumes data. These representative examples show the possibilities enabled by an integrative VaaS with a reusable and distributed data storage API.

The functions in Listing 6.2 define a reusable library of code that gets included with each request to Weaver. The specific code in this listing is what defines the translation from arbitrary latitude, longitude, and pressure floating point values to quantized integer coordinates that identify a voxel.

The code in Listing 6.3 is an example of the code that gets executed to compute new flow lines, and then post-processed (using the library functions in Listing 6.2) to be used as a graph. It is important to note that this code is idiomatic Lua code and that its main purposes are: organizing data into a request for other VaaS, and post-processing that response for the frontend interface.

6.4 Weaver: Usability Study

The results of the preceeding studies (Sections 3.4, 4.4, 5.4) focused primarily on scalability. For Weaver, although scalability is important, the primary goal is to look towards the future at what is possible with VaaS. As a result, this section will explore the expressiveness and versatility of the Weaver system.

In terms of purpose, Weaver is a VaaS that demonstrates integrating many other VaaS together. As a result, the majority of its code is code to integrate services together. This means that the amount of interfacing code is a direct measurement of the ease for integrating services. The interfacing code is server-side code that is written once by a VaaS developer, and then used many times as a user.

The secondary point of measurement for Weaver is in the amount of code necessary to make requests to the VaaS. This is user-written code that gets sent with each request. The goal of this code is to be small and simple enough to achieve the metaphor of "weaving" different VaaS together.

The interfacing code for each of the VaaS integrations is small. For integrating with the parallel flow VaaS (Chapter 3), there is around 20 lines of bridging code on the server-side. Of these lines, half of them are for validating inputs with sensible error messages, half are for generating sensible output, and only two lines of code are for directly calling the low-level flow processor. This trend of 20 lines of interfacing code holds for both the data-management (Chapter 4) and graph visualization (Chapter 5) interfaces.

The frontend code is the last area of importance for evaluating Weaver's abilities. Weaver is intended to be idiomatic for the ObservableHQ notebook platform. For my purpose, this means that the code should be simple and well organized because it is intended to be used and modified by other developers. To that end, Weaver's primary frontend interface is through a JavaScript tagged template literal, which is common on ObservableHQ for embedding different languages with a JavaScript-focused environment. This allows Weaver code to remain idiomatic for its platform, enabling simple integrations with other libraries and tools within the ObservableHQ platform.

Chapter 7

Conclusion

Data visualization continues to be a key tool for the modern scientist. With the compounding effects of remote work becoming more common while data becomes more massive in scale, the move towards Visualization-as-a-Service is imminent. I believe that the "as-a-Service" paradigm will continue to grow and become an important part of every developer and scientist's toolchains.

This dissertation has presented several advancements to the design and implementation of VaaS that make VaaS more powerful, expressive, and versatile for more applications. These improvements aim to address challenges with moving infrastructure to a service architecture.

7.1 Data-Parallel Visualization-as-a-Service

I have shown (Chapter 3) that by incorporating stateful information about the distribution of data resources across a service, it is possible to make a VaaS for an application domain that previously has to solely rely on traditional HPC. Specifically as a driving example, I have adapted the design of parallel-flow visualization to work on shared cloud resources, utilizing the elastic scalability that clouds provide. As a result, the parallel-flow service is capable of providing on-demand access to TB's of 3D turbulent flow data.

The benefits of this VaaS flow visualization service include: 1) immediate on-demand accessibility, 2) flexible interactivity requiring only an internet connection, and 3) pay-as-you-go cost affordability. These benefits are especially important for improving the ease of

access to community data repositories. In essence, a data repository can become more than just a place to download datasets, but also serve as a way for immediate experimentation with datasets for those without expert-level knowledge of working with scientific data.

7.2 Multi-Device Visualization-as-a-Service

The success of Alpaca (Chapter 4) shows the promise of coordinating multiple devices together for the same visualization, especially for the pairing of physical spaces, surfaces, and 3D mixed-reality interfaces. The reusable core design of Alpaca as a object-store with stateful event listeners enables flexibility across different applications and use-cases. A guiding principle behind Alpaca is its intent to upgrade applications unintrusively and without changes to the original application's code.

I am excited by the prospects of applying this core design to different domains by bridging multiple applications and devices with minimal code changes. In Chapter 6, I have shown that the same premise can be used for coordinating different resources of different types in the same way as Alpaca coordinated different interfaces. Specifically, Alpaca's model enables the coordination of particle flow visualization with large graph visualization, with minimal linking code between the two.

7.3 Hypothesis-Oriented Visualization-as-a-Service

I have demonstrated a novel method for visualizing large graphs (Chapter 5) using Graph Shaders (GS) capable of rendering graphs with 100's of millions of edges. This visualization is efficient and capable of round-trip interaction times under 1 second. The success of GS can be largely attributed to its design for representing rendering requests as small and intrinsicly parallel programs.

The design of GS suggests that when the visualization needs of a user require more than a singular stateless request, then one can consider using a domain specific language (DSL) as the interface for VaaS. This philosophy is similar to all other "system" areas of computer science.

In Chapter 6, I have demonstrated how this concept of representing DSL programs within requests can be used to transform a simplistic set of independent services, for parallel flow visualization and graph visualization, into a single integrated service.

Bibliography

[npm] Node package manager. https://npmjs.com/. Accessed: 2020-05-14. 86, 102

[w3t] Usage Statistics of JavaScript as Client-Side Programming Language on Websites. https://w3techs.com/technologies/details/cp-javascript. Accessed: 2020-02-02. 70

[dom] W3C Document Object Model. https://www.w3.org/DOM/. Accessed: 2020-02-02. 63

[4] (2018 (accessed October 14, 2018)a). Software Container Platform - Docker: https://www.docker.com/. 7, 20

[5] (2018 (accessed October 14, 2018)b). Software Container Platform - Docker: https://www.docker.com/. 87

[6] (2019). http://d3js.org/. 18

[7] Abello, J., Van Ham, F., and Krishnan, N. (2006). Ask-graphview: A large scale graph visualization system. *IEEE transactions on visualization and computer graphics*, 12(5):669–676. 14

[8] Adai, A. T., Date, S. V., Wieland, S., and Marcotte, E. M. (2004). Lgl: creating a map of protein function with an algorithm for visualizing very large biological networks. *Journal of molecular biology*, 340(1):179–190. 14, 102

[9] Ahn, S., Ko, H., and Yoo, B. (2014). Webizing mobile augmented reality content. *New Review of Hypermedia and Multimedia*, 20(1):79–100. 12

[10] Aigner, W., Miksch, S., Schumann, H., and Tominski, C. (2011). *Visualization of time-oriented data*. Springer Science & Business Media. 91

[11] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. (2013). Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044. 25

[12] Andrienko, N. and Andrienko, G. (2006). *Exploratory analysis of spatial and temporal data: a systematic approach*. Springer Science & Business Media. 91, 92

[13] Ayachit, U. (2015). The Paraview guide: a parallel visualization application. 6

[14] Bach, B., Sicat, R., Beyer, J., Cordeil, M., and Pfister, H. (2017). The Hologram in My Hand: How Effective is Interactive Exploration of 3D Visualizations in Immersive Tangible Augmented Reality? *IEEE Transactions on Visualization and Computer Graphics*. 11, 12

[15] Badam, S. K., Fisher, E., and Elmqvist, N. (2015). Munin: A peer-to-peer middleware for ubiquitous analytics and visualization spaces. *IEEE Transactions on Visualization and Computer Graphics*, 21(2):215–228. 12

[16] Bastian, M., Heymann, S., and Jacomy, M. (2009). Gephi: An open source software for exploring and manipulating networks. 13

[17] Beck, F., Burch, M., Diehl, S., and Weiskopf, D. (2017). A taxonomy and survey of dynamic graph visualization. In *Computer Graphics Forum*, volume 36, pages 133–159. Wiley Online Library. 14

[18] Beck, M. (2019a). On the hourglass model. *Communications of the ACM*, 62(7):48–57. 5, 7

[19] Beck, M. (2019b). On the hourglass model. *Commun. ACM*, 62(7):48–57. 88

[20] Belcher, D., Billinghurst, M., Hayes, S. E., and Stiles, R. (2003). Using augmented reality for visualizing complex graphs in three dimensions. *Proceedings - 2nd IEEE and ACM International Symposium on Mixed and Augmented Reality, ISMAR 2003*, pages 84–93. 11

[21] Bikakis, N., Liagouris, J., Krommyda, M., Papastefanatos, G., and Sellis, T. (2016). Graphvizdb: A scalable platform for interactive large graph visualization. In *2016 IEEE 32nd international conference on data engineering (ICDE)*, pages 1342–1345. IEEE. 14

[22] Billinghurst, M., Kato, H., and Poupyrev, I. (2001a). The magicbook-moving seamlessly between reality and virtuality. *IEEE Computer Graphics and applications*, 21(3):6–8. 64

[23] Billinghurst, M., Kato, H., and Poupyrev, I. (2001b). The magicbook-A Transitional AR Interace. *Computer Graphics and Applications*, 21(3):6–8. 11

[24] Binyahib, R., Pugmire, D., Yenpure, A., and Childs, H. (2020). Parallel particle advection bake-off for scientific visualization workloads. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 381–391. IEEE. 52

[25] Björk, S., Holmquist, L. E., and Redström, J. (1999). A framework for focus+context visualization. In *Proceedings of the 1999 IEEE Symposium on Information Visualization*, INFOVIS '99, pages 53–, Washington, DC, USA. IEEE Computer Society. 14

[26] Bostock, M., Ogievetsky, V., and Heer, J. (2011). D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309. 37, 41

[27] Brehmer, M., Sedlmair, M., Ingram, S., and Munzner, T. (2014). Visualizing dimensionally-reduced data: Interviews with analysts and a characterization of task sequences. In *Proceedings of the Fifth Workshop on Beyond Time and Errors: Novel Evaluation Methods for Visualization*, BELIV '14, pages 1–8, New York, NY, USA. ACM. 67

[28] Chau, D. H., Kittur, A., Hong, J. I., and Faloutsos, C. (2011). Apolo: making sense of large network data by combining rich user interaction and machine learning. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 167–176. ACM. 14

[29] Chen, C.-M. and Shen, H.-W. (2013). Graph-based seed scheduling for out-of-core ftle and pathline computation. In *2013 IEEE symposium on large-scale data analysis and visualization (LDAV)*, pages 15–23. IEEE. 25

[30] Childs, H., Brugger, E., Whitlock, B., Meredith, J., Ahern, S., Pugmire, D., Biagas, K., Miller, M., Harrison, C., Weber, G. H., Krishnan, H., Fogal, T., Sanderson, A., Garth, C.,

Bethel, E. W., Camp, D., Rübel, O., Durant, M., Favre, J. M., and Navrátil, P. (2012). VisIt: An end-user tool for visualizing and analyzing very large data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pages 357–372. 6

[31] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113. 58, 88

[32] Doleisch, H., Gasser, M., and Hauser, H. (2003). Interactive feature specification for focus+context visualization of complex simulation data. In *Proceedings of the Symposium on Data Visualisation 2003*, VISSYM '03, pages 239–248, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association. 14

[33] Doyle, R. P., Chase, J. S., Asad, O. M., Jin, W., and Vahdat, A. (2003). Model-based resource provisioning in a web service utility. In *USENIX Symposium on Internet Technologies and Systems*, volume 4, pages 5–5. 52

[34] Easterling, D. R., Meehl, G. A., Parmesan, C., Changnon, S. A., Karl, T. R., and Mearns, L. O. (2000). Climate extremes: observations, modeling, and impacts. *science*, 289(5487):2068–2074. 39

[35] Ellson, J., Gansner, E. R., Koutsofios, E., North, S. C., and Woodhull, G. (2004). Graphviz and dynagraph—static and dynamic graph drawing tools. In *Graph drawing software*, pages 127–148. Springer. 13

[36] Emeras, J., Varrette, S., Plugaru, V., and Bouvry, P. (2016). Amazon elastic compute cloud (ec2) vs. in-house hpc platform: a cost analysis. *IEEE Transactions on Cloud Computing*. 18

[37] Fang, D., Keezer, M., Williams, J., Kulkarni, K., Pienta, R., and Chau, D. H. (2017). Carina: Interactive million-node graph visualization using web browser technologies. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 775–776. International World Wide Web Conferences Steering Committee. 14

[38] Fruchterman, T. M. and Reingold, E. M. (1991). Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164. 14

[39] Fu, M., Agrawal, A., Floratou, A., Graham, B., Jorgensen, A., Li, M., Lu, N., Ramasamy, K., Rao, S., and Wang, C. (2017). Twitter heron: Towards extensible streaming engines. In *2017 IEEE 33rd international conference on data engineering (ICDE)*, pages 1165–1172. IEEE. 25

[40] Gajdoš, P., Ježowicz, T., Uher, V., and Dohnálek, P. (2016). A parallel fruchterman–reingold algorithm optimized for fast visualization of large graphs and swarms of data. *Swarm and Evolutionary Computation*, 26:56–63. 14

[41] Geipel, M. M. (2007). Self-organization applied to dynamic network layout. *International Journal of Modern Physics C*, 18(10):1537–1549. 14, 102

[42] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. (2014). Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA. USENIX Association. 13

[43] Grubert, J., Langlotz, T., Zollmann, S., and Regenbrecht, H. (2016). Towards pervasive augmented reality: Context-awareness in augmented reality. *IEEE transactions on visualization and computer graphics*, 23(6):1706–1724. 12

[44] Hall, B. H. and Adam, B. (2002). The NBER Patent-Citations Data File: Lessons, Insights, and Methodological Tools. *Patents, citations, and innovations: A window on the knowledge economy*, page 403. 86, 102

[45] Hampel, T., Bopp, T., and Hinn, R. (2006). A peer-to-peer architecture for massive multiplayer online games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06, New York, NY, USA. ACM. 12

[46] Hariri, S., Tunc, C., and Badr, Y. (2017). Resilient dynamic data driven application systems as a service (rdaas): A design overview. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 352–356. IEEE. 7

[Heyman] Heyman, S. Google books article on the new york times. https://www.nytimes.com/2015/10/29/arts/international/google-books-a-complex-and-controversial-experiment.html. Accessed: 2018-08-15. 70

[48] Hill, A., MacIntyre, B., Gandy, M., Davidson, B., and Rouzati, H. (2010). Kharma: An open kml/html architecture for mobile augmented reality applications. In *2010 IEEE International Symposium on Mixed and Augmented Reality*, pages 233–234. IEEE. 12

[49] Hobson, T., Duncan, J., Raji, M., Lu, A., and Huang, J. (2020). Alpaca: AR graphics extensions for web applications. In *Proc. of IEEE VR (accepted)*. 56

[50] Hobson, T. and et al. (2021). Interactive visualization of large turbulent flow as a cloud service. *IEEE Transactions on Cloud Computing*, pages 1–1. 16, 87, 88

[51] Hota, A. (2019). Vaas: Visualization as a service. 9

[52] Hu, Y. (2005). Efficient, high-quality force-directed graph drawing. *Mathematica Journal*, 10(1):37–71. 14, 102

[53] Hu, Y. (2011). Algorithms for visualizing large networks. *Combinatorial Scientific Computing*, 5(3):180–186. 14

[54] Inness, P. M. and Slingo, J. M. (2003). Simulation of the madden–julian oscillation in a coupled general circulation model. part i: Comparison with observations and an atmosphere-only gcm. *Journal of Climate*, 16(3):345–364. 39

[55] Irawati, S., Hong, S., Kim, J., and Ko, H. (2008). 3D Edutainment Environment : Learning Physics through VR / AR Experiences. *Science And Technology*, pages 21–24. 11

[56] Jackson, K. R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., Wasserman, H. J., and Wright, N. J. (2010). Performance analysis of high performance computing applications on the amazon web services cloud. In *2010 IEEE second*

international conference on cloud computing technology and science, pages 159–168. IEEE. 6

[57] Jacomy, M., Venturini, T., Heymann, S., and Bastian, M. (2014). Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PloS one*, 9(6). 102

[58] Jeuken, A., Siegmund, P., Heijboer, L., Feichter, J., and Bengtsson, L. (1996). On the potential of assimilating meteorological analyses in a global climate model for the purpose of model validation. *Journal of Geophysical Research: Atmospheres*, 101(D12):16939–16950. 39

[59] Jiang, M., Van Essen, B., Harrison, C., and Gokhale, M. (2014). Multi-threaded streamline tracing for data-intensive architectures. In *2014 IEEE 4th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 11–18. IEEE. 25

[60] Kendall, W., Wang, J., Allen, M., Peterka, T., Huang, J., and Erickson, D. (2011). Simplified parallel domain traversal. In *SC11: Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 10:1–10:11. 25

[61] Kerracher, N., Kennedy, J., and Chalmers, K. (2015). A task taxonomy for temporal graph visualisation. *IEEE transactions on visualization and computer graphics*, 21(10):1160–1172. 91, 92

[62] Krichenbauer, M., Yamamoto, G., Taketomi, T., Sandor, C., and Kato, H. (2017). Augmented Reality vs Virtual Reality for 3D Object Manipulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(8):1–1. 12

[63] Kurtzer, G. M., Sochat, V., and Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5). 7

[64] Leach, P., Michael, M., and Salz, R. (2005). RFC 4122: A Universally Unique Identifier (UUID) URN namespace. RFC. 30

[65] Lee, B., Plaisant, C., Parr, C. S., Fekete, J.-D., and Henry, N. (2006). Task taxonomy for graph visualization. In *Proceedings of the 2006 AVI workshop on BEyond time and errors: novel evaluation methods for information visualization*, pages 1–5. 91, 92

[66] Lee, G. A., Yang, U., Kim, Y., Jo, D., Kim, K.-H., Kim, J. H., and Choi, J. S. (2009). Freeze-set-go interaction method for handheld mobile augmented reality environments. In *Proceedings of the 16th ACM Symposium on Virtual Reality Software and Technology*, pages 143–146. ACM. 64

[67] Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data. 86, 102

[68] Leskovec, J. and Sosič, R. (2016). Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1. 14

[69] Martin, S., Brown, W. M., Klavans, R., and Boyack, K. W. (2011). Openord: an open-source toolbox for large graph layout. In *Visualization and Data Analysis 2011*, volume 7868, page 786806. International Society for Optics and Photonics. 102

[70] Meinshausen, M., Raper, S. C., and Wigley, T. M. (2011). Emulating coupled atmosphere-ocean and carbon cycle models with a simpler model, magicc6-part 1: Model description and calibration. 39

[71] Minocha, S. and Hardy, C. L. (2011). Designing navigation and wayfinding in 3d virtual learning spaces. In *Proceedings of the 23rd Australian Computer-Human Interaction Conference*, pages 211–220. ACM. 11, 12

[72] Müller, C., Camp, D., Hentschel, B., and Garth, C. (2013). Distributed parallel particle advection using work requesting. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pages 1–6. IEEE. 25

[73] Munzner, T. (1998). Drawing large graphs with h3viewer and site manager. In *International Symposium on Graph Drawing*, pages 384–393. Springer. 14

[74] Murray, N. (2011). Contextual interaction support in 3d worlds. In *Distributed Simulation and Real Time Applications (DS-RT), 2011 IEEE/ACM 15th International Symposium on*, pages 58–63. IEEE. 11, 12

[75] Nachmanson, L., Prutkin, R., Lee, B., Riche, N. H., Holroyd, A. E., and Chen, X. (2015). Graphmaps: Browsing large graphs as interactive maps. In *International Symposium on Graph Drawing*, pages 3–15. Springer. 14

[76] Noghabi, S. A., Paramasivam, K., Pan, Y., Ramesh, N., Bringhurst, J., Gupta, I., and Campbell, R. H. (2017). Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645. 25

[77] Oviatt, S. and Cohen, P. (2000). Multimodal interfaces that process what comes naturally. *Communications of the ACM*, 43:45–53. 11, 12

[78] Pai, V. S., Druschel, P., and Zwaenepoel, W. (1999). Flash: An efficient and portable web server. In *USENIX Annual Technical Conference, General Track*, pages 199–212. 52

[79] Pascucci, V., Scorzelli, G., Summa, B., Bremer, P.-T., Gyulassy, A., Christensen, C., Philip, S., and Kumar, S. (2012). The visus visualization framework. *EW Bethel, HC (LBNL), and CH (UofU), editors, High Performance Visualization: Enabling Extreme-Scale Scientific Insight, Chapman and Hall/CRC Computational Science*. 6

[80] Peixoto, T. P. (2014). The graph-tool python library. *figshare*. 13

[81] Peterka, T., Ross, R., Nouanesengsy, B., Lee, T.-Y., Shen, H.-W., Kendall, W., and Huang, J. (2011). A study of parallel particle tracing for steady-state and time-varying flow fields. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 580–591. IEEE. 25

[82] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1988). Numerical Recipes in C. 32

[83] Pugmire, D., Childs, H., Garth, C., Ahern, S., and Weber, G. H. (2009). Scalable computation of streamlines on very large datasets. In *Proceedings of the 2009 ACM/IEEE conference on Supercomputing*. 25

[84] Pugmire, D., Kress, J., Chen, J., Childs, H., Choi, J., Ganyushin, D., Geveci, B., Kim, M., Klasky, S., Liang, X., et al. (2020). Visualization as a service for scientific data. In *Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI: 17th Smoky Mountains Computational Sciences and Engineering Conference, SMC 2020, Oak Ridge, TN, USA, August 26-28, 2020, Revised Selected Papers 17*, pages 157–174. Springer. 8

[85] Rabbi, I. and Ullah, S. (2013). A survey on augmented reality challenges and tracking. *Acta graphica: znanstveni časopis za tiskarstvo i grafičke komunikacije*, 24(1-2):29–46. 64

[86] Raji, M., Hota, A., Hobson, T., and Huang, J. (2018a). Scientific visualization as a microservice. *IEEE Transactions on Visualization and Computer Graphics (accepted)*. 7, 8, 87, 88, 122

[87] Raji, M., Hota, A., Hobson, T., and Huang, J. (2018b). Scientific visualization as a microservice. *IEEE transactions on visualization and computer graphics*. 58

[88] Raji, M., Hota, A., and Huang, J. (2017a). Scalable web-embedded volume rendering. In *IEEE Symp. on Large Data Analysis and Visualization (LDAV)*, pages 45–54. 7

[89] Raji, M., Hota, A., and Huang, J. (2017b). Scalable web-embedded volume rendering. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 45–54. IEEE. 58

[90] Raji, M., Hota, A., and Huang, J. (2017c). Scalable web-embedded volume rendering. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 45–54. 87, 88, 122

[91] Raji, M., Hota, A., Sisneros, R., Messmer, P., and Huang, J. (2017d). Photo-guided exploration of volume data features. *arXiv preprint arXiv:1710.06815*. 7

[92] Robinson, J. T., Turner, D., Durand, N. C., Thorvaldsdóttir, H., Mesirov, J. P., and Aiden, E. L. (2018). Juicebox. js provides a cloud-based visualization system for hi-c data. *Cell systems*, 6(2):256–258. 8

[93] Rogelj, J., Meinshausen, M., and Knutti, R. (2012). Global warming under old and new scenarios using ipcc climate sensitivity range estimates. *Nature climate change*, 2(4):248–253. 39

[94] Rouzati, H., Cruiz, L., and MacIntyre, B. (2013). Unified webgl/css scene-graph and application to ar. In *Proceedings of the 18th International Conference on 3D Web Technology*, pages 210–210. ACM. 12

[95] Saad, L. and Wigert, B. (2021). Remote work persisting and trending permanent. 4

[96] Saha, S., Moorthi, S., Wu, X., Wang, J., Nadiga, S., Tripp, P., Behringer, D., Hou, Y.-T., ya Chuang, H., Iredell, M., Ek, M., Meng, J., Yang, R., Mendez, M. P., van den Dool, H., Zhang, Q., Wang, W., Chen, M., and Becker, E. (2011). NCEP Climate Forecast System Version 2 (CFSv2) 6-Hourly Products. 16, 18, 43

[97] Satyanarayan, A., Moritz, D., Wongsuphasawat, K., and Heer, J. (2016a). Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics*, 23(1):341–350. 14, 37

[98] Satyanarayan, A., Russell, R., Hoffswell, J., and Heer, J. (2016b). Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE transactions on visualization and computer graphics*, 22(1):659–668. 14

[Schmuhl] Schmuhl, M. graphopt. https://web.archive.org/web/20220611030748/http://www.schmuhl.org/graphopt/. Accessed: 2023-06-23. 102

[100] Shannon, P., Markiel, A., Ozier, O., Baliga, N. S., Wang, J. T., Ramage, D., Amin, N., Schwikowski, B., and Ideker, T. (2003). Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research*, 13(11):2498–2504. 13

[101] Shen, H.-W., Hansen, C. D., Livnat, Y., and Johnson, C. R. (1996). Isosurfacing in span space with utmost efficiency (issue). In *Proceedings of Seventh Annual IEEE Visualization'96*, pages 287–294. IEEE. 108

[102] Shneiderman, B. (1996). The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages*, pages 336–343. 14

[103] Stasko, J. and Zhang, E. (2000). Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *IEEE Symposium on Information Visualization 2000. INFOVIS 2000. Proceedings*, pages 57–65. 14

[104] Stonebraker, M., Çetintemel, U., and Zdonik, S. (2005). The 8 requirements of real-time stream processing. *ACM Sigmod Record*, 34(4):42–47. 25

[105] Szalavári, Z. and Gervautz, M. (1997). The personal interaction panel–a two-handed interface for augmented reality. In *Computer graphics forum*, volume 16, pages C335–C346. Wiley Online Library. 11

[106] Tanahashi, Y., Chen, C.-K., Marchesin, S., and Ma, K.-L. (2010). An interface design for future cloud-based visualization services. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 609–613. IEEE. 9

[107] van der Maaten, L. and Hinton, G. (2008). Visualizing data using t-sne. *J. of Machine Learning Research*, 9:2579–2605. 14

[108] Wetzlmaier, T., Ramler, R., and Putschögl, W. (2016). A framework for monkey gui testing. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 416–423. IEEE. 43

[109] Wood, J., Brodlie, K., Seo, J., Duke, D., and Walton, J. (2008). A web services architecture for visualization. In *2008 IEEE Fourth International Conference on eScience*, pages 1–7. IEEE. 7

[110] Yi, J. S., ah Kang, Y., and Stasko, J. (2007). Toward a deeper understanding of the role of interaction in information visualization. *IEEE transactions on visualization and computer graphics*, 13(6):1224–1231. 5, 67

[111] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., et al. (2016). Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65. 25

Vita

Tanner Hobson received a BS in Computer Science from the University of Tennessee, Knoxville (UTK) in 2017. While earning their BS, they had several internships that focused on data science, scientific visualization, and user interface design: in 2014 at Oak Ridge National Laboratory (ORNL) under Dr. Laura Pullum and Dr. Arvind Ramanathan; in 2015-2017 at UTK under Dr. Jian Huang; and in 2016 at ORNL under Dr. Ricardo Leal. In 2017, Tanner began work on a PhD in Computer Science and continued to gain real world experience: in 2020-2021 at Argonne National Laboratory (ANL) under Dr. Orcun Yildiz, Dr. Bogdan Nicolae, and Dr. Tom Peterka; and in 2021-2023 at Intel Federal under Jim Jeffers and Dr. Johannes Günther. Tanner successfully defended their dissertation on June 19th, 2023.