Rapid Development of a Data Visualization Service in an Emergency Response

Saiful Khan, Phong H. Nguyen, Alfie Abdul-Rahman, *Member, IEEE,*
Euan Freeman, Cagatay Turkay, and Min Chen, *Member, IEEE*

Abstract—We present the design and development of a data visualization service (RAMPVIS) in response to the urgent need to support epidemiological modeling workflows during the COVID-19 pandemic. Facing a set of demanding requirements and several practical challenges, our small team of volunteers had to rely on existing knowledge and components of services computing, while thinking on our feet in configuring services composition and adopting suitable approaches to services engineering. Through developing the RAMPVIS service, we have gained useful experience of ensuring conformation to services computing standards, enabling rapid development and early deployment, and facilitating effective and efficient maintenance and operation with limited resources. This experience can be valuable to the ongoing effort for combating the COVID-19 pandemic, and provides a blueprint for visualization service development when future needs for visual analytics arise during emergency response.

Index Terms—Web services, services computing, service composition, services engineering, data visualization, epidemiological modeling, emergency response, REST, ontology, agents, open source, template-based development, rapid deployment, RAMPVIS.

1 Introduction

Services engineering is concerned with the development, maintenance, and operation of service-oriented systems [1], [2]. Developing a service-oriented system involves a software life-cycle that typically takes tens of months. In ordinary circumstances, it is perfectly reasonable to have a relatively long life-cycle to ensure the quality of the delivered service. However, in some circumstances, such as an emergency response, it may be necessary to develop a service-oriented system in a few weeks or months. During the COVID-19 pandemic, we encountered the need to provide a group of epidemiologists and modeling scientists with a data visualization service. The urgency of this necessitated a process for progressive development, a schedule for rapid deployment, and mechanisms for effective and efficient maintenance and operation.

In March 2020, the Royal Society (UK) issued a nationwide call for skilled researchers to join the *Rapid Assistance in Modelling the Pandemic (RAMP)* initiative [3]. This led to the establishment of several large consortia of volunteers, among which was the *Scottish COVID-19 Response Consortium* (SCRC) [4]. On 14 May 2020, one SCRC coordinator met a visualization scientist who answered the RAMP call, and concluded that they needed more data visualization specialists to support the pandemic response. Next day, a call for visualization volunteers was sent out. By the end of May, over 20 visualization researchers and developers joined the effort to provide visualization support to the epidemiological modeling workflows in the SCRC.

When the visualization volunteers formed the RAMPVIS group [5], [6], several teams of epidemiologists and modeling scientists in the SCRC were working intensively on multiple epidemiological models, while a team of research software engineers were developing the SCRC data infrastructure to host a huge volume of data, including captured COVID-19 data, model predictions, parameters and internal data of simulation runs, and so on. One urgent requirement was therefore to enable users to visualize data hosted by the SCRC data infrastructure, which was expected to be ready and operational in two to three months. In order to not disrupt the ongoing development of the SCRC data infrastructure at that time, the RAMPVIS group decided to develop a visualization service to be coupled with the SCRC data infrastructure. By the end of August 2020, the SCRC data infrastructure was operational. Within a few weeks the visualization service, called RAMPVIS, was deployed, providing numerous plots and dashboards and other features to support the domain experts' visual analytics needs.

We could not easily follow a traditional software engineering workflow for developing a visualization service because of several key challenges, including the urgency in an emergency, the lack of stable programming resources due to the volunteer nature of our activities, and the lack of an existing visualization service infrastructure to help bootstrap the development. Facing a set of demanding requirements and several practical challenges, we had to rely on existing knowledge and system components of services computing as much as possible, while thinking on our feet in configuring service composition and choosing appropriate approaches for service engineering. Through the development of the RAMPVIS service, we have gained useful experience of ensuring conformation to service com-

- *Saiful Khan and Min Chen are with University of Oxford. E-mail: saiful.khan@eng.ox.ac.uk, min.chen@oerc.ox.ac.uk.*
- *Phong H. Nguyen is with Redsift Ltd. and University of Oxford. E-mail: phong.nguyen@redsift.io.*
- *Alfie Abdul-Rahman is with King's College London. E-mail: alfie.abdulrahman@kcl.ac.uk.*
- *Euan Freeman is with University of Glasgow. E-mail: euan.freeman@glasgow.ac.uk.*
- *Cagatay Turkay is with University of Warwick. E-mail: cagatay.turkay@warwick.ac.uk.*

Manuscript received April 19, 2005; revised August 26, 2015.

puting standards, enabling rapid development and early deployment, and facilitating effective and efficient maintenance and operation. We believe that this experience can be valuable to the ongoing effort for combating the COVID-19 pandemic (and its ever-changing needs for visual analytics).

The many activities of the RAMPVIS consortium have been summarized in a report [6] that gives a high-level overview of ongoing work across several visualization teams and offers little insight into the technical infrastructure developed during this project. In a recent paper [7], we described a novel technique for generating visualizations and dashboards semi-automatically. That work focuses on a specific algorithmic approach and user interface for assuring visualizations are propagated to appropriate data. In this article, we focus more on the novel technical aspects of the RAMPVIS service architecture and our service development processes, which were absent in those earlier works. Our contribution includes reflections on our experience of developing a data visualization service in the emergency response to the COVID-19 pandemic. We also present our service design and our approach to its design, engineering, maintenance, operation, and continual improvement. Our general approach, open source software contributions, and lessons learned will also be relevant to future service computing initiatives during emergency response efforts.

2 Related Work

2.1 Rapid Development in Emergency Response

Our volunteer consortium were tasked with developing a COVID-19 visualization service for epidemiologists and modeling scientists. As an emergency response, we adopted a rapid development approach focused on timely service delivery, in a way that made the best use of our volunteers' expertise and skills. There is a large body of work reflecting on agile/rapid development approaches and, more recently, discussions on the use of such approaches in software development during COVID-19 can be found in the literature. When starting, we considered a variety of rapid development approaches. No one software development process or set of practices is universally and ideally suited to all contexts, and the constraints of each setting need to be considered when choosing a suitable approach [8].

Extreme programming (XP) has been adopted for the rapid development of web applications [9] and emergency response system development [10]. Some XP characteristics and practices were suited to our situation: e.g., fewer than ten team members, use rapid prototyping, continuous integration, simple design, refactoring, small release cycles, etc. However, practices like pair programming, onsite support, test-driven development, planning, etc., were not feasible in an emergency situation. Base agile methods (e.g., Scrum, Kanban, and flow) and large-scale agile methods (e.g., SAFe and Scrum-at-scale) were not wholly ideal due to factors largely influenced by our volunteer effort, e.g., lack of a scrum master, time constraints needed to properly define and document project scope, and uncertain and limited time commitments from volunteers working remotely and at different times. Adopting such a process takes time and involves solving different challenges at different adoption stages [11]. As this paper goes on to show, rapid development in a volunteer-driven emergency response requires drawing on different engineering practices, rather than wholly committing to one.

This can also be seen in other discussions of COVID-19 related software initiatives, although these works do not cover their approach or reflect on their experiences in much detail. Krausz et al. [12] reflect on the rapid development of a COVID-19 crisis management system (e.g., for patient intake, monitoring, referral) in Ontario, Canada. In developing a centralized web service, they were able to quickly put in place a system that allowed the health service to take a data-driven approach to their pandemic approach. Schinköthe et al. [13] similarly described the rapid development of a web application supporting telehealth delivery, motivated by the urgent need to adopt remote health service provision to protect front-line health professionals from COVID-19 transmission. In discussing their system, they reflect on the benefits the agile approach brought to the fast deployment of this service. Beyond software, others have reflected on the benefits of agile development approaches in, e.g., medical device development [14]. These works lack technical contributions and offer limited insight into service design and engineering practice. Our work addresses this gap with a more technical oriented contribution about service engineering during (and for) the COVID-19 response.

Rapid development has also been adopted in visualization deployment, most relevant to our work. For example, Dixit et al. [15] described the rapid development of a system for visualizing telehealth data, so that hospitals could take data-informed approaches to dealing with the sudden increase in telehealth service provision as a result of COVID-19 disrupting typical healthcare practice. In reflecting on their work, they discussed the importance of rapid development in providing visualizations to inform healthcare provision – noting that "basic visualization is better than no visualization" in emergency response. Our project had the more challenging scope of providing a COVID-19 visualization service that could flexibly support emerging analytics needs and scale across thousands of diverse data types and sources. Our contribution thus focuses on the unique services engineering challenges of developing such a scalable visualization system, rather than one with a more narrowly-defined scope and feature set like [15].

There are existing commercial web-based visualization platforms (e.g., Tableau, PowerBI) which could be used to support visualization service development and have seen some uptake by local and national governments for visualizing key indicators in their COVID-19 data (e.g., Public Health Scotland's COVID-19 dashboard built on Tableau [16]). However, these platforms are too costly for a volunteer effort like ours (c.f., [7]) and their architecture is not conducive to developing a flexible and scalable visualization service that would quickly meet the needs of the SCRC epidemiologists and modeling scientists. Instead, we used a rapid development approach based around templates and ontology-supported development leading to a visualization service that could be quickly scaled and extended to deal with the rapidly-evolving visual analytics needs of our users.

2.2 Template-based Rapid Web Development

Template-based approaches are often used in web application development, reducing the amount of work necessary to develop and deploy functional systems and services. Popular examples include Django and the Jinja template engine. Such development approaches can increase developer productivity, provide uniformity, and present a common look and feel across web applications [17]. Template approaches are compelling because they can increase efficiency in rapid development [18] and reduce the need for infrastructure implementation [19] – this was critical in our context, given the timely need to develop a bespoke visualization service, especially with the limited volunteer resources available.

Template approaches also support a clear separation of concerns [20], [21], which can be especially beneficial in urgent volunteer efforts like RAMPVIS. For example, some of our volunteers had expertise in data processing, whilst others specialized in visualization, epidemiological modeling, etc. Our general use of a template approach meant we could decouple aspects of service component development to make the best use of our volunteers and their areas of expertise; e.g., so that visualization experts could focus on visualization implementation without being exposed to other components and the underlying data infrastructure.

2.3 Ontology-supported Development

An ontology is a structured representation of knowledge. These can be valuable in software engineering because they can be used to formally represent software entities, domain knowledge, and the relationships between them. Reviews of ontology-supported software engineering give insight into the benefits of this practice [22], [23], [24]. These structured knowledge representations not only support implementation, but can help with long-term maintenance [25], software re-use [26], documentation [27], integration of semantic features [28], and software testing [29].

Ontologies have also been used in service computing, e.g., to support service specification [30] and service composition [31] through structured representations of service components and associated domain knowledge. We used an ontology as a core component in our visualization service: the ontology represented the relationships between key service components (visualization implementations, data streams, and interactive web pages). This enabled us to rapidly scale the system through semi-automatic visualization production [7]. Similar to template-based development, we argue that ontologies can be used to rapidly scale the functional offerings of a service with minimal development resource; in our context, this was critical in ensuring COVID-19 visualizations were quickly made available with, e.g., complete coverage of UK health board areas.

3 Service Requirements and Methodology

The RAMPVIS service was developed in response to the growing need to visualize and gain analytical insight into a huge volume of COVID-19 data, hosted by the SCRC data infrastructure and, later, other data providers. This data infrastructure was designed to support several epidemiological modeling teams in the consortium, each with their own data and visualization needs. The overall requirements for the service were immediately apparent, including:

1) *To conform to services computing standards*: The SCRC, since its formation, had an overarching policy of open data and open source software. This led to the decision to utilize widely-available web technology to deliver data visualization to the epidemiologists and modeling scientists in the SCRC. Consequently, RAMPVIS needed to conform to the standards for developing and delivering services via the web.
2) *To be developed rapidly and deployed as early as possible*: In an emergency response, time is without doubt a critical factor. While the development of the SCRC data infrastructure commenced just before the RAMPVIS service, there was a pressing requirement for synchronizing their deployment, so that domain experts could quickly access data and corresponding visual analytics capabilities. This led to an ambitious plan of making both the data infrastructure and the RAMPVIS service available before the autumn of 2020. Since the RAMPVIS group was formed at the end of May, there were only a few months for designing and engineering a deployable data visualization service.
3) *To be maintained and operated effectively and efficiently*: Most of our visualization volunteers were university faculty members and could not commit a sufficient and persistent amount of time for software engineering. The responsibility for developing the RAMPVIS service rested on four members of the *generic support team*, including two industrial researchers and two faculty members. They were supported by three other faculty members who assisted in architecture and user interface design, in liaison with other activities in the SCRC and the RAMPVIS group. Because of the unpredictable nature of a volunteering development operation [32], and the potentially long period needed to combating the COVID-19 pandemic, it was necessary to ensure that the RAMPVIS service would be easy to maintain and operate, and could be resilient to changing personnel for engineering, maintaining, operating, and redeveloping the service.

We also faced the challenge of having no previous work on visualization services in the literature, which would have otherwise provided a development 'template' to kick-start development. Brodlie et al. reviewed a range of visualization services developed two decades ago [33]; at that time, many visualization techniques (e.g., volume rendering, flow visualization, and virtual reality) demanded significant computational resources. Much of the effort then was to address the need for high-performance computation, with services offering infrastructure for costly visualizations. In recent years, these visual designs are readily available on everyday devices and contemporary visualization tools (e.g., D3.js) typically do not require infrastructural support. Reports on infrastructure-based visualization became rare in the literature. We therefore drew on the knowledge of service composition and engineering from other web applications, and adapted such knowledge to design our visualization service composition and inform our approach to service engineering.

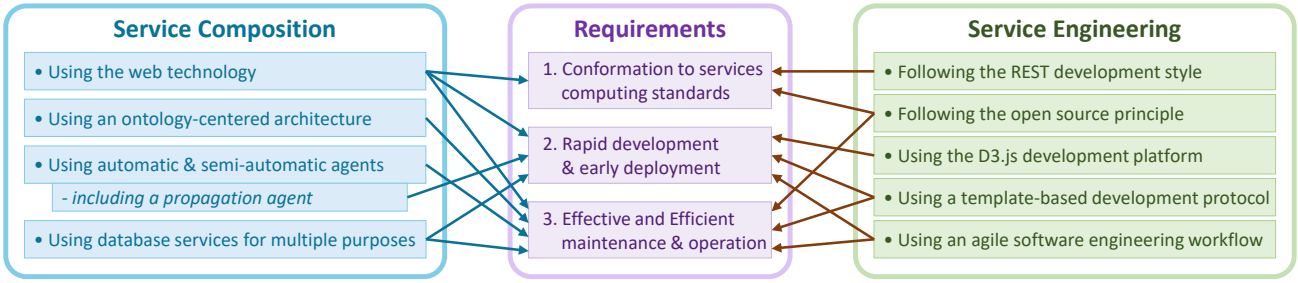


Fig. 1: A summary of the service components and engineering approaches that were designed to meet the requirements for developing a visualization service in response to the emergency.

To address the aforementioned requirements and challenges, we made the following design choices:

- *Developing for web*: This would allow users to access RAMPVIS via their web browser, simplifying deployment across multiple platforms. This would also accelerate service development because (i) we could use generic UI functionality offered by web browsers, so we could focus on application-specific UI components, and (ii) we could use D3.js, a web-based visualization platform popular in the visualization community.
- *Using an ontology-centered architecture*: Epidemiological modeling workflows use many pieces of data (called *data streams* in the RAMPVIS service). A visualization plot or a dashboard (called *VIS functions*) may display multiple data streams and can be applied to many other data streams as well. When a VIS function is combined with a set of data streams, it becomes a *web page*. It is challenging to maintain knowledge about relationships among data streams, VIS functions, and web pages as distinct system components. We chose to use an ontology to represent such knowledge in a structured way, with the ontology being a central component in the RAMPVIS service that could support development, facilitate automation features, and, later, support scalability.
- *Using automatic and semi-automatic agents*: Most COVID-19 data normally changes on a daily basis, while modeling data changes whenever a model is executed. We thus developed a number of automatic and semi-automatic agents to deal with dynamic changes in data streams. These significantly reduced maintenance and operation costs, while some (e.g., the propagation agent) facilitated an effective and efficient process for deploying and scaling the RAMPVIS service.
- *Using database services for multiple purposes*: Our service would be data-intensive and needed to store both application data and metadata (e.g., about COVID-19 data streams), as well as operational data. Database services were used for many components in the RAMPVIS service, so that data storage and processing components were quick to develop and easy to maintain.
- *Using the D3.js development platform*: D3.js was the most familiar visualization platform in the volunteer team. We thus used D3.js to: (i) accelerate the service design and engineering process, (ii) maximize the development resources and experiences so a service could be delivered quickly, and (iii) benefit from open source visualizations written for D3.js. This platform was compatible with key requirements (e.g., web-based).
- *Using a template-based development protocol and a propagation agent*: Based on the limited estimated volunteering time (seven person months from June–December 2020), we quickly formulated a template-based development protocol for programming a visualization plot or a dashboard. This allowed visualization developers to focus on implementing individual visualization templates (*VIS functions*) with little distraction from underlying service infrastructure. Since a VIS function could be applied to many data streams after creation (e.g., thousands of time series in the SCRC data infrastructure), we developed a propagation agent for connecting VIS functions to appropriate data streams, based on the ontology [7]. The template-based development protocol and the propagation agents facilitated rapid development and scaling of the RAMPVIS service, and simplified maintenance of the relationships between numerous VIS functions and associated data streams.
- *Using an agile software engineering workflow*: An agile approach was ideal for the development life-cycle of RAMPVIS, which involves collaboration (i.e., with the SCRC), continual requirements discovery (i.e., for epidemiological modeling), adaptive planning (e.g., according to the ongoing development of the SCRC data infrastructure), early delivery (i.e., in an emergency response), evolutionary development, and continual improvement. It allowed us to respond flexibly to changes in user requirements, development resources (volunteering and funded), and most of all, the evolving situation of the pandemic.
- *Following open source principles*: All RAMPVIS components would be developed as open-source software and code is available via GitHub [34], [35], [36]. This conforms to the SCRC requirement for openness and has helped with: team communication and coordination; software testing, deployment, and update; and training new developers after we received grant funding.

The above design choices affect both *service composition* and *service engineering*, as illustrated in Fig. 1. In the remainder of this paper, we discuss each of these in turn.

4 Services Composition

Configuring a service by selecting appropriate technical components and their composition is a key task in services computing. We will discuss the architecture of the

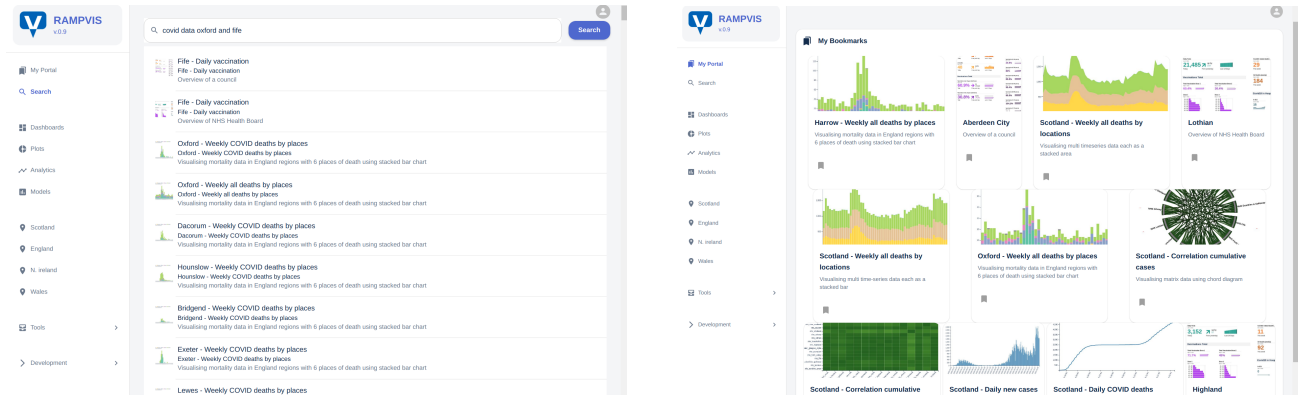


Fig. 2: Examples of the search (left) and portal (right) facilities of the RAMPVIS service. These facilities display plots and dashboards as thumbnails that are generated automatically by a *ThumbnailAgent* that will be described in Section 4.3.3.

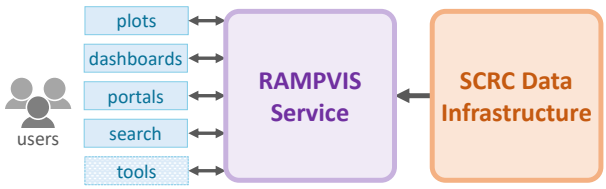


Fig. 3: The functional role of the RAMPVIS service.

RAMPVIS service (Section 4.1) as configured to satisfy the requirements and design choices outlined earlier. We then describe two purposely-designed aspects of our service composition: the ontology for knowledge management (Section 4.2), and (semi-)automatic agents (Section 4.3). Finally, we will highlight other important design features in configuring the RAMPVIS service.

As shown in Fig. 3, the essential functional role of the RAMPVIS service is to transfer data from a data infrastructure to numerous plots and dashboards, in order to meet the visual analytics needs of its users. There is a huge (and constantly growing) volume of data and thousands of plots and dashboards created to meet emerging user needs, so it is necessary to provide users with search and personalization features for quick access to the visualizations that are most relevant to, and thus frequently used by, each individual (as shown in Fig. 2). The RAMPVIS service is currently being extended to provide more complex visualization facilities within self-contained subsystems, referred to as "tools". These are the key components from which our service is composed.

4.1 RESTful Architecture

Representational state transfer (REST) is a popular software architectural style for web-based services [37]. We adopted the Richardson Maturity Model [38] and CoHA Maturity Model [39] in designing the RAMPVIS service as a RESTful architecture, and in selecting APIs for facilitating operations and communications in response to client requests.

The RAMPVIS service back-end is a tiered architecture [37], [40], [41], where the service operations are grouped into five tiers (T0—T4) as illustrated in Fig. 4.

T0 is a web server and its load-balancer that receives incoming requests via HTTP and distributes these to available instances of our REST APIs.

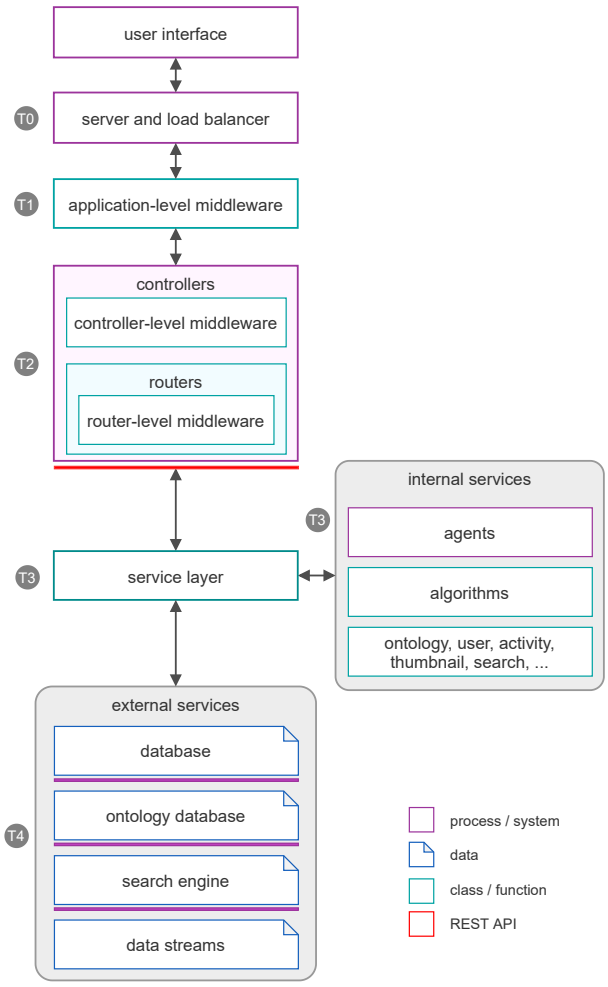


Fig. 4: An illustration of our n-tier architecture and functionality of each tier. A server and its load-balancer (T0) receives the incoming requests through HTTP and forward the requests to our REST APIs. In REST API layer, application-level middlewares (T1) first process the request and forward successful requests to controllers and route-handlers (T2). Finally, the routed requests are resolved by service layer (T3) using internal and external services (T4).

T1 hosts the application-level middleware that first processes a request and then forwards it to controller(s).

T2 hosts controllers, routers and other lower-layer middle-

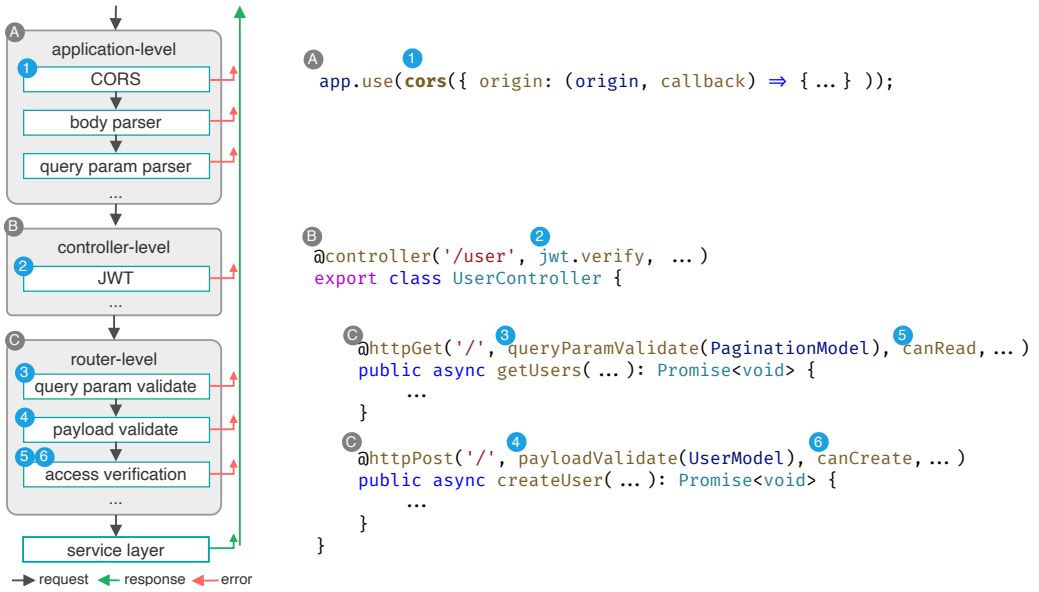


Fig. 5: An illustration of request processing and handling (e.g., parse, sanitize, verify, and validate) by middleware using a flowchart (left) and code snippets (right). At application-level (A), the CORS (1) middleware allows only requests from our approved list of origins (e.g., https://vis.scrc.uk). At controller-level (B), a JSON Web Token verification middleware (2) checks if the requests are from an authenticated users. At router-level (C), (3) checks if the request has valid pagination query parameter, (5) checks if the user has read permission, The route (D) handles creation of users, where we verify if the request payload has valid user data (4) and the user has valid access right (6) for creating a user in our system (user with admin role has such right).

ware.

T3 is a service layer that acts on a request, and in turn calls other services to resolve it. For example, for a login request, the service layer will call a *UserService* to authenticate the login request using a database that keeps authentication information.

T4 provides a connection to all the physically isolated external services.

T1 and T2 provide the REST API logic, while T3 and T4 provide various microservices. Each tier can be logically and/or physically separated. Here, the logical separation means that the software classes and functions in different tiers may run in a same container, while physical separation means that independent instances of software stacks must run in different containers.

In our services composition, T0 and external services in T4 are physically separated, running in their own containers. This is because these services are mostly existing third-party systems and they are easier and more reliable to maintain when they are physically isolated. T1, T2, and T3, which were developed by us, are logically separated but running in a same container. Because a container (e.g., Docker) packs up a service or software stack and all its dependencies, T1, T2, and T3 can run quickly and reliably.

We introduced middleware for initial processing of the requests, and controller classes for logically similar resources and endpoints (discussed later).

Middleware is a regular feature in services computing, providing functions in handling HTTP requests, typically for prepossessing, security checking, sanitizing, validation before passing them to the service layer. In our services composition, we have three middleware levels in T1 and T2, with "global" functions for preprocessing of all requests, "groupwise" functions for preprocessing logically-similar requests, and specialized functions for preprocessing individual requests.

As illustrated in Fig. 5, the three middleware levels are *application*, *controller*, and *router*. The application-level provides "global middleware" functions. For example, a *CORS* (Cross-Origin Resource Sharing) filters out requests from unapproved domains (see also Fig. 5(A)); a *Body Parser* parses requests with JSON payloads; a *Query Param Parser* parses requests with URL-encoded query parameters; and so on. The requests that pass the application-level middleware are forwarded to a relevant controller in the controller-level middleware.

Each *controller* is defined as a class that groups and implements logically similar routes. For example, all user routes are implemented in a *UserController* (see also Fig. 5(B)). *JWT* (JSON Web Token) checks if a request contains a valid authentication token (see also Fig. 5(2)). The requests that passes the controller-level middleware are forwarded to a relevant router.

A *router* is a route-handler. For example, the *Query Param Validate* middleware function validates the query parameters of a request (see also Fig. 5(3)) and the *Access Verification* function checks if a request has an appropriate role-based access right (see also Fig. 5(5,6)).

By conforming to the REST architectural style, we were be able to implement T1, T2, and T3 with modular and maintainable code, while utilizing as many existing commercial services in T0 and T4. This enabled us to speed up the development by using open-source frameworks and libraries, and freeware services and systems. In particular, we used NGINX server and load-balancer in T0, and MongoDB database and Elasticsearch search engine in T4.

We implemented our REST API using state-of-the-art frameworks FastAPI and Node.js, and our middleware using the Express.js library. Other RAMPVIS components (described later) implemented CPU-intensive services (e.g., most agents) using Python, and I/O intensive operations (e.g., database, ontology, search engine, visualization) using asynchronous JavaScript.

According to the taxonomy of services composition proposed by [42], the RAMPVIS service features the following techniques and standards:

- Language:
  - Component:
    - Type: Data, Application Logic
    - Interaction Protocol: REST
    - Description: Swagger
    - Data Format: JSON
    - Interaction Style: Push, Pull
    - Selection: Runtime
  - Target Application: *Emergency Responses*
  - Notation: Textual: Code-based
  - Paradigm: Script-based
  - Composite Constructs: Control Flow Patterns: Sequence, Parallel
  - Crosscutting Concerns:Exceptions
- Knowledge Reuse:
  - Reused Artifacts: Components
  - Reused Technique: Copy/Paste, Cloning
- Automation: *Agents* (see Section 4.3)
- Tool Support: Refactoring
- Execution Platform:
  - Deployment Options: Cloud
  - Execution Engine: Service Bus
- Target User: End-user Programmer: App Developers

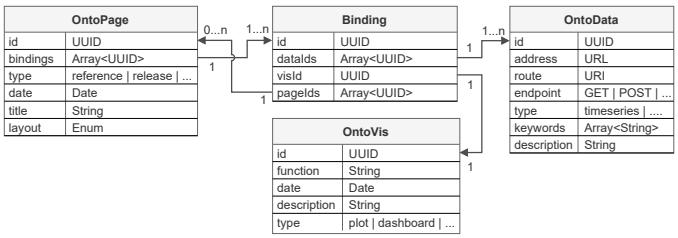


Fig. 6: A document-based data model or schema of our ontology. We assign a universally unique identifier (UUID) to each object and use the *id* field to define relationships between objects.

4.2 Ontology for Knowledge Management

We used an ontology to manage the complex knowledge about the different datasets, visualization functions, and web pages that make up the RAMPVIS service. This plays a central role in our service, as it represents the composition of its key semantic components. Fig. 6 gives an overview of the ontology and its four main classes: **OntoData** (data), **OntoVis** (visualizations), **Binding** (a mapping of data streams to a visualization function), and **OntoPage** (web pages resulting from such a binding).

Each basic data entity is referred to as a *data stream*, which is a data object that can be passed to a visualization function as a single data structure. For example, the most common types of data streams in the RAMPVIS service are time series and matrices. These are represented by the **OntoData** class, which holds data stream metadata as a list of attributes, including: the *address* of the data server; *route* and *endpoint* of REST APIs to access the data; data *type* (e.g., time series, cumulative time series); and a set of *keywords* that describe the data. These objects are created and updated by download agents (Section 4.3) when a data stream arrives from the SCRC data infrastructure.

Each basic visualization (i.e., an instance of a plot or dashboard) is referred to as a *VIS function*, which is a function that receives one or more data streams as inputs, and renders a visual representation on a web-page. These visual representations can be as simple as a bar chart displaying a single data streams, or as complex as a dashboard displaying multiple data streams. These are represented by the **OntoVis** class, which holds visualization metadata attributes, including: the *name* (e.g., *StackedBarChart*), the *type* of data streams it visualizes (e.g., time series, cumulative time series, matrix), and *function* name for execution.

When a visualization is mapped to data stream(s), it results in a web page that users can access via the RAMPVIS service. Visualizations can be applied to many similar groups of data streams: e.g., a visualization for one region can be applied across all other regions with similar data. Likewise, the same data stream can be mapped to other visualizations to provide different views for the same data. Hence, there are numerous potential bindings of data streams to visualization functions. These are represented by the **Binding** class, which records a one-to-many relationship between **OntoVis** and **OntoData** objects.

The **OntoPage** class corresponds to a web page served up by the RAMPVIS service, presenting **Bindings** in a way that users can access and interact with via their web browser. Our service uses a template-based approach (described later) to facilitate this.

Complex visualizations functions (e.g., dashboards showing numerous smaller plots or individual data values) may feature hyperlinks to other **OntoPage** objects (i.e., web pages for each individual plot or corresponding data stream). For example, a user may click a key indicator on a dashboard, leading to a different web page that displays the whole time series corresponding to that data stream. In each object of the **Binding** class, there is thus an attribute *pageIds* that holds the identifiers of all linked **OntoPage** objects.

This ontology is a crucial part of our service architecture and its development was a key technical achievement in our volunteer effort. As discussed in a visualization paper by Khan et al. [7], this ontology facilitates the rapid deployment of a vast number of visual designs with minimal volunteer effort, helping us rapidly scale our visual analytic offering. Our ontology-based architecture (and its open source implementation) can be used as a core service component in future visualization infrastructures, especially those used in settings where large-scale visual analytics are needed quickly and at low-cost.

4.3 Automatic and Semi-automatic Agents

The RAMPVIS service makes use of several *agents*, i.e., software components that act autonomously towards an

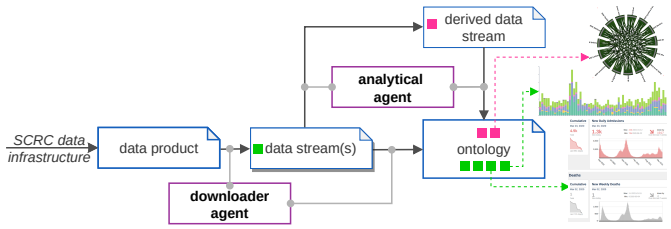


Fig. 7: Downloader and Analytical agents make dynamic data and derived metrics (as derived data streams) ready for visualizations.

objective [43], [44]. These run independently from the main service processes and their key objectives are to perform repetitive and frequently needed tasks, e.g., pre-computing complex models or data values, batch processing information, and automating time-consuming tasks. In Fig. 4, agents are shown as an 'internal service'. Fig. 7 shows how two agents, in particular, relate to key components in our service architecture. In this section, we outline four agents that reduced the need for volunteer resources, helped scale our service as its complexity grew, and supported the addition of complex features. This is not an exhaustive list of agents in our service; e.g., later in the paper we discuss the *Propagation* agent underlying our template-based engineering approach.

4.3.1 Downloader Agent for Automatic Data Updating

Our service was initially built to visualize the large collection of *data products* in the SCRC data infrastructure. Here, each data product typically contains many related data streams. It would not be sensible to pass a whole data product to a visualization function, e.g., if only a subset are to be visualized, so it was necessary to break products down to their constituent data streams.

We developed a *Downloader* agent to facilitate this. This agent maintains a list of known data streams, together with their updating schedule, corresponding data products, and the data query specification. Its objective is to ensure associated data and metadata is kept up to date, providing the individual data streams for our service. Since these agents run periodically, we used the *Advanced Python Scheduler* (*APScheduler* library) for scheduling these processes. This agent can easily be adapted to support other data sources and provides a layer of abstraction between our ontology and external data sources. Its key benefit to our service is introduced automation that means volunteer effort is only needed to register new data sources.

At the scheduled times for each data stream, the *Downloader* agent sends a pre-defined query to the data provider, receives the data stream, and updates the local data store and ontology service components. For example, when the time series of positive cases in the 14 Scottish regions were updated (daily), the *Downloader* agent would automatically extract the time series from the relevant data product and replace the 14 corresponding data streams in our service.

4.3.2 Analytical Agents for Data Processing

It was often necessary to apply analytical algorithms to process raw data streams, e.g., to compute derived values or to update epidemiological models. For example, consider the need to compare time series data of positive COVID-19 cases across the 14 regions of Scotland. There are many similarity/difference measures in the literature and some are computationally expensive. These computations can be offloaded to a software agent, so that values can be pre-computed instead of in real-time. *Analytical* agents were executed after corresponding *Downloader* agents finished their tasks, meaning that all relevant visualizations would thereafter show the updated data (and derived values) without human intervention.

Ongoing work is extending the use of these capabilities to add new analytics features and support more complex visual analytics features, e.g., to introduce analytical agents for uncertainty analysis. The *Analytical* agent framework provides a solid foundation for introducing these more advanced capabilities and gives a general means of supporting new tools and functionalities, which are now being introduced after obtaining funding for dedicated research software engineers.

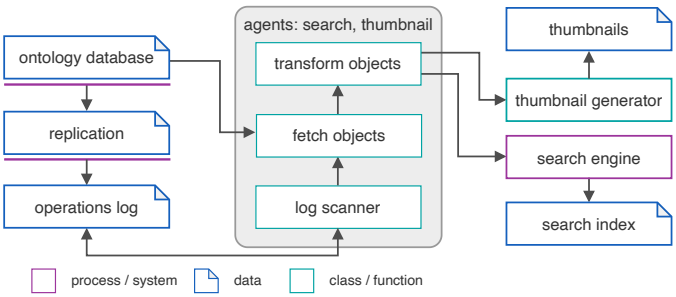


Fig. 8: Every operation in the ontology database is logged in an operations log file. A log scanner periodically scans the log file, and if there is any ontology operation (e.g., create, update, and delete) on a page, it retrieves the *OntoPage* object from the ontology database. The ontology object is sent to our thumbnail agent to generate thumbnail of the page and search engine to update search index and make the page search-able.

4.3.3 Thumbnail Agent for Generating Images

Some parts of the RAMPVIS end-user interface required thumbnails of plots and dashboards, e.g., within search results and user portals. These thumbnails were generated from 'live' data rather than generic representative images– to increase relevance and utility to our users. As such, we introduced a *Thumbnail* agent to produce these updated thumbnail images, avoiding costly in-time generation (e.g., when providing dozens of search results).

The *Thumbnail* agent is scheduled to update thumbnails regularly. As shown in Fig. 8, it scans the operation logs regularly and detects new operations; when a new web page is created or an existing web page is updated, it fetches the web page, its linked VIS function, and the relevant data streams. Using the page *id*, the thumbnail generator formulates a URL and opens the URL in a headless browser using the APIs provided by Selenium and Puppeteer; when the headless browser renders the page, the *Thumbnail* agent takes a screenshot. This is then cropped and downsized to a thumbnail, which is stored in the service database.

4.3.4 Search Index Agent

A search index is an inverted-index data structure that consists of: (i) a list of unique words related to an object or document; and (ii) for each word, a list of objects in which it appears. We implemented search functionalities so our users could search and discover COVID-19 visualizations available in our service. We used the Elasticsearch search engine and implemented an agent that periodically updates the search index with web page information in the ontology database. Search indexing occurs when relevant changes are detected in our operations logs; for example, when new web pages are modified or created (e.g., via the propagation service). The index incorporates information from OntoPage, OntoData, and OntoVis objects, allowing complex search queries, e.g., with data descriptions, keywords, and visualization types.

5 Services Engineering

As discussed in Sections 1, 2, and 3, it would be a challenge to develop a data visualization service in a normal circumstance since there has not been much reported about such infrastructure-level services in the literature. Developing such a service in an emergency response with volunteering effort was a non-trivial undertaking. In this Section, we describe our approach to address the requirements in Section 3 from the perspective of services engineering. In particular, we describe a template-based workflow, in conjunction with a propagation agent, for creating visualization software and content in Section 5.1. We then describe our adoption of an agile software life-cycle for developing the RAMPVIS service in Section 5.2.

5.1 Template-based Visualization Development

Templates are an efficient way to implement and render dynamic data into web pages; e.g., because we can reuse a template when the underlying data model is consistent. Templates also provide uniformity and a common look and feel for all web pages; e.g., we can reuse a base web page structure (as in Fig. 9(1)). The ontology has thousands of OntoPage objects generated through propagation. We adapted the model-view-controller (MVC) [45] design pattern where the application data (i.e., OntoPage objects) represents the model, and the template implements the view layer that renders the model. The template defines dynamic elements using variables and expressions, replaced with model attributes when the template is rendered to a web page. The process is illustrated in Fig. 9.

Developers use web template systems (with varying degrees of success) to maintain this separation [20]. Our approach separated the implementation of the view layer and model layer. In addition to that, separation of visual design logic from the user interface or template was possible through a visualization function factory (Fig. 9(11)). Our visualization volunteers specialized in JavaScript visualization programming libraries, e.g., D3.js, however, had limited knowledge of user interface, template, and services development. In addition, the presentation variations in templates are content-invariant, meaning a template developer can update the presentation without broader infrastructural implications. Hence, our design helped us to collaborate effectively and use the volunteer resources efficiently, with each volunteer contributing based on their own expertise.

In this emergency response, we had to remain agile regarding which templating system to use. Several templating systems are available based on *when* templates replace placeholders with actual content and render/assemble web pages. For example, in a server-side system, the run-time substitution happens on the web server; in a client-side system, run-time substitution happens in the web browser; in the outside server-based system, the static web pages are produced offline and uploaded to the webserver, and so on. Initially, we used a plain HTML template; then we implemented Python Jinja-based templates (the syntax used in Fig. 9), which use a client-side rendering system. We substituted the Jinja templates with React.js templates, which support both client-side and server-side rendering. Finally, we incorporated Next.js, which helped us generate static web pages offline and upload them to the webserver. This was a dynamic process, in response to changing implementation needs as our service scaled, and new user and developer needs emerged. In the next section, we discuss the reasons for this evolution. The visualization design logic remained unchanged, however, as any templating engine can execute visualization designs using the function factory.

5.1.1 Automatic Propagation of Visual Designs

When we started working with COVID-19 data, we had thousands of time series representing somewhat similar data sets. When one visualization researcher designed a plot (e.g., visualizing COVID-19 related deaths in one region of the UK), the same design could be reused for many regions (e.g., the 14 and 336 regions in Scotland and England, respectively). When we bind data from one region to a suitable visual design (i.e., reference binding), we can reuse the same design to visualize data from other regions. We developed a propagation technique that used the reference binding to discover potentially related groups of data streams and semi-automatically propagate the visual design to them. By decoupling the data and visualization components and managing their relationships via the ontology, this was straightforward and allowed our visualization service to rapidly scale its visual analytics offerings, with minimal volunteer effort. A detailed description of propagation technique and quality assurance interface can be found in [7].

As described before, an OntoPage may relate to many Binding(s), producing a web page with many visual designs in a general use case. It will be time-consuming to develop an algorithm to discover and generate $1 : 1..n$ binding automatically. In our propagation work, we simplified it to use only a $1 : 1$ binding. The propagation process generated thousands of OntoPage objects semi-automatically that are stored in the ontology database, and a reusable template will process those to produce visualization web pages.

5.2 Agile Service Engineering

In May 2020 when our volunteer initiative started, our key priority was to rapidly deliver an operational visualization infrastructure to the SCRC modeling scientists and epidemiologists. Our initial challenges were limited volunteer resources, constantly evolving requirements, and the urgency

```
...
1 {% extends "base-site.html" %}
...
<h3>
2   {{ ontoPage.title | safe }}
</h3>
...
3 <div id="chart"></div>
...
<script>
4   let ontoPage = {{ input | tojson }};
5   let bindings = {{ ontoPage.bindings | safe }};
...
  $(document).ready(function() {
...
6     let args = {
7       placeholder: $('#chart'),
8       data: await service.fetchData(binding.dataIds),
        links: await service.getLinks(binding.pageIds)
      };
9     let type = await service.getVis(binding.visId);
10    VisFunctionFactory.get(type, args);
  });
...
```

```
11 class VisFunctionFactory {
     static get(type, args) {
...
12     if (type === "StackedBarChart") {
         return new StackedBarChart(args);
       }
...
     }
...
   }
```

```
13 class StackedBarChart {
...
    constructor(args) {
...
14    let data = args.data;
15    let links = args.links;
16    let selector = args.placeholder;
      let svg = d3.select(selector)
                  .append("svg");
      this.draw(svg, data, links);
    }
...
17  draw(svg, data, links) {
...
    }
...
  }
```

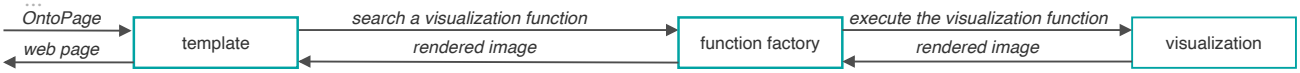


Fig. 9: A template takes an OntoPage object as an input, de-serializes it to a JSON object (4), and replaces template variables, e.g., title (2). We use the binding (5) attributes to fetch data (7), links (8), and visualization function details (9). We implemented a function factory (10) using the factory design pattern [45]. The factory when called (10) resolves the appropriate visualization function, for example, a *StackedBarChart* (11). The visualization function (13) extracts data (14), links (15), and CSS selector (16) information from function argument. The placeholder is where the actual plot or dashboard will be drawn as an SVG (3). Each visualization function has a *draw* function (17) implementing the visualization rendering.

of contributing to this emergency response. We also faced the challenges of the dynamic nature of COVID-19 data (e.g., the emergence of new key indicators, new data types, etc). Responding to new user needs and acting on their feedback were also important factors. These challenges contributed to our choice of an agile methodology. In software development, when the base process is the V-Model or the Waterfall, everything is considered *traditional* and when the process is connected to Scrum or Kanban is considered *agile*, regardless of whether the process of interest is objectively best choice for the respective context [46]. A large-scale empirical study found that only a small percentage (15%) of projects use purely traditional or agile in their projects [8]. Our approach was more of *being agile*. Being agile practice has a more substantial impact on the perception of agility than methods suited for the dynamically changing situation [8].

At the beginning we had only few volunteers: one with systems engineering expertise, three with visualization design and development expertise, and three who worked with domain experts to coordinate with architecture development. We introduced a *separation of concern* design pattern in our architecture for dividing the system into logical layers of functionality (e.g., data, services, user interface, and visualization designs). This separation enabled volunteers to contribute independently to their areas of expertise.

At that point, we explored state-of-the-art industrial-scale solutions (e.g., Angular, Electron, React.js) and libraries for user interface development, then we developed small prototypes (v.0.1). We evaluated these, including evaluation of how visualizations would be developed. Post analysis, we agreed that this approach was not suitable, as the resources had limited time to learn advanced frameworks, which would delay the initial development significantly. We decided to use HTML templates instead to produce web pages with visualizations. The template-based approach helped us maintain consistency in the user interface and across all visualizations, rather than free-form generation. Within the user interface, we provided a separation of concerns using the *factory pattern* to make templates and visualization designs independent and reusable (see Section 5.1). We knew that the plain-HTML templates would not be scalable and maintainable, and we may need to replace the template layer as more advanced user interface functionalities are required. By June 2020, we had produced 70+ plots and two dashboards. In the meantime, the STFC commissioned required VMs and servers for deployment. We deployed our visualization infrastructure so it was available to users. This way, we started to adopt monthly release cycles which are still continuing.

By July 2020, we started receiving a wider range of COVID-19 related data, more dynamic and inconsistent from others, from different regions of the UK. We re-purposed visual designs, e.g., plots and dashboards. We noticed that the same datastreams sometimes appeared in many visualizations, e.g., once in individual plots and again in a dashboard providing overviews of related key indicators. We realized that we needed to store this knowledge (i.e., the mappings between visualizations and data sets) in a better way, so the same plots did not need to be recreated. An ontology was added to address this. Developing a scalable ontology is time-consuming, so we first recorded the knowledge in a JSON file, which would later be migrated to a scalable ontology database.

As we created more plots and dashboards to broaden our visual analytics offering, we found it challenging to manage the growing number of resulting HTML pages. At this stage we migrated our code-base to use advanced and reusable templating libraries like Jinja. Using this new templating approach, we generated 100+ plots, four dashboards, and few analytical visualizations with reduced boilerplate implementation. Since visualization components were decoupled from this, the technical aspects of developing visualizations

remained unchanged; instead, the changes were at the user interface layer - i.e., constructing the pages served to users.

Several visualizations require metrics computed from raw data and some computations were expensive. To improve throughput, we decided to pre-compute analytics instead of calculating on demand. We created an agent to perform the computations periodically (Section 4.3.2), and we deployed that as a minor release in September.

The quantity of data started to grow exponentially towards the end of 2020 as the SCRC data infrastructure rapidly increased its offering. Leading up to this period, we focused on scaling our ontology. We explored different data models for the ontology, e.g., RDF [47], OWL [48], property graph [49], [50], and document model (e.g., MongoDB, CouchDB, etc.). We also investigated databases such as Neo4J and MongoDB by writing mini prototypes. We decided to implementing the ontology in MongoDB. Although this is an unconventional approach for modeling and implementing ontology, it was a good fit for our service. MongoDB allows applications to store virtually any structure [51], provide enough free and managed cloud instances, e.g., MongoDB Atlas, and client drivers for rapid prototyping using declarative syntax. By the end of December, we completed our design to scale the ontology to support the increased volumes of data while keeping the system efficient. We developed a wrapper using an *adapter pattern* to seamlessly use our earlier JSON-based ontology, until we migrated it to the new scalable ontology database.

In January 2021, we introduced a download agent to automate the downloading and registration of data with the ontology, both helping to deal with the increasing volume of data now available to the visualization service, and taking advantage of the structured representation of visualization and data entities within the ontology. In February, we developed an ontology management UI using Angular.

In March, we developed propagation algorithms, propagation user interface, and quality assurance for propagation. By April, we had already received 4500+ data streams. We were able to create 700+ plots and 45+ dashboards using the propagation technique, using that structured knowledge representation to semi-automatically produce these to offer scale. In May 2021, we refactored our Python REST API migrating from Flask to FastAPI. This supported extended development as FastAPI provides typing, faster implementation of data verification and validation, and implementation of dependency injection design patterns.

In June 2021, we started to analyze requirements for integrating advanced visualization tool support (e.g, ensemble visualization, and timeseries similarity search and visualization). Research funding had led to team growth by this point, with new Research Software Engineers (RSEs) working on these advanced features.

In July 2021, we migrated our Jinja-based template and user interface to the React.js stack. This required refactoring our visualization function code, but the benefit of this migration was that our user interface became more efficient and easier to maintain. We began with simple technologies and progressively migrated to scalable solutions; this supported efficient use of resources, as agile recommends. We added new scheduler agents to index search data in the backend, to speed up the search process in real time.

The infrastructure was developed and tested with the SCRC data product as a primary source. After deployment, we also registered and visualized data from many sources (e.g., ONS and Our World in Data), which demonstrates that our system is robust and agnostic to data source. We also performed an end-to-end drill to clone and deploy our visualization infrastructure in a separate server provided by the STFC. The entire process took two hours in total. We documented the steps for future use. This proved that the same system can be redeployed successfully and could be re-purposed for different causes.

In August 2021, we implemented server-side rendering using Next.js framework. A thumbnail agent was added so user can refer to thumbnail of the visualizations in search and portal page. An improved page search technique was also added, which will also support implementation of multi-faceted search in future.

5.3 Reflection

Our deliveries and work exhibit the core ideas of agile manifesto. A team of volunteers working remotely would not have been able to rapidly implement a tightly coupled monolithic architecture in an agile way. In contrast to the monolithic, a services architectural pattern allowed us to decouple our system into smaller, independent services, allowing us to adopt agile principles. We structured our system around these services and autonomously implemented them the various platforms that our volunteers were familiar with. The services adopted REST principles for seamless composition and integration.

There are tradeoffs between basic and advanced frameworks, libraries, and approaches; in turn, these have implications for volunteer-based development efforts. Basic frameworks help rapid development but require a degree of a boilerplate code for implementing scalable solutions. In contrast, advanced frameworks provide the necessary boilerplate code and aid scaling; however, these requires significant learning effort and time, delaying initial release cycles. We refactored our underlying frameworks and approaches throughout the development to achieve a scalable solution; in retrospect, this was not an efficient use of time and future efforts can learn from this mistake. Our work can be used as a blueprint for bootstrapping and rapidly developing a scalable data visualization system for any application domain. We proposed required methodologies, e.g., a tiered architecture, composition of different external and internal services, implementation guidelines and open-source code for reusable and modular services, and services for template-based data visualization.

In retrospect we realized, due to fluid requirements, our system required continuous changes, and changes to architecture required much attention and often introduced bugs. The most time-consuming challenges we encountered was testing, debugging, and profiling individual services and the system composed of many services. We did not write unit and regression tests; we also did not specify testing contracts that should have been clearly defined between services developed by individuals. These sometimes created problems while integrating the services in production. We recognize that in an emergency-response scenario it is appealing to

focus on functional and user-facing development tasks, but our experience was a humbling reminder that testing and quality assurance can save time in the long run.

There are systems available for supporting diagnosis of services using log monitoring, exception or error stack-tracing, and passive tracing [52] through network traces. Investigation of those logs requires substantial cognitive efforts by the developers, which takes time away from feature development. There are opportunities for the service computing community to develop more easily adaptable libraries or zero-configuration tools for capturing, collecting, and storing, e.g., services logs, performance metrics (e.g., execution time and memory), and services request processing traces at a granular level. Likewise, there are opportunities for the visualization community to develop approaches for intuitive visual analytics of captured metrics that can help quickly diagnose, troubleshoot, and profile service behavior and performance.

6 Discussion

Developing a data visualization service as a team of volunteers during an emergency situation introduced unique challenges, most importantly the time-pressure and limited development resources available. These constraints meant that we had to make choices to get us to a minimally viable product as quickly as possible in the early days of development, although these decisions might not always be the most effective and robust in the long term. For instance, we started using technologies that are most familiar for the volunteer developers such as Jinja for template-based development, even though it is not one of the most high-performing or the popular of libraries. As soon as a functional minimal solution has been reached and the volunteer team had more availability to focus on performance enhancements, these technologies were later replaced by more efficient libraries, such as React.js and Next.js.

Similar choices were made in approaching the code base development and programming design as well. We gave more prominence to the regular refactoring of existing code in the earlier stages of the project to diversify the functionality quickly, while we paid less attention to perfecting the design or the re-usability of the code. As the development matured, however, we focused more on re-usability and good design practices while coding.

We have significantly benefited from, and built on, various open source projects and libraries in our development. Such open source libraries are not only supporting the rapid and effective development of the various components, but also the knowledge base and continual support by their respective communities are critical in resolving issues rapidly. A further benefit of incorporating open source libraries, especially those with lively and active communities, is that the libraries are maintained and developed further continuously, which is critical in maintaining the operation of emergency response services with little resources for the upkeep of the services. A further criteria when choosing appropriate open source libraries and frameworks has been their suitability to modular development, as well as their proven scalability since we expect our services to be able to handle many data streams, thousands of visualizations and dashboards, and many users effectively.

Our tiered architecture design also enabled us to more effectively allocate our limited development resources to achieve a more streamlined development process. A dedicated developer focused on the design and development of the middleware and external services and the ontology, another developer focused on developing agents and diversifying the analytical capabilities offered by the agents, while the visualization developers could focus only on developing the visualizations without worrying about the technical aspects of data and fit within the framework [7].

7 Conclusion

In this paper, we described the design and development of a data visualization service that was created in response to an urgent need for supporting epidemiological modeling workflows during the COVID-19 pandemic. Our service architecture requirements and our development approach were both heavily influenced by this time-critical situation, and the need to effectively and flexibly provide visualizations of large (and growing) sets of data. We discussed how a series of service components and service engineering methods emerged as a viable, scalable, and flexible solution to the those requirements and challenges. The RAMPVIS service is a successful outcome of this, providing a visualization infrastructure that has evolved to meet ever-changing visual analytics needs throughout the pandemic.

Our efforts and the resulting data visualization service demonstrate that it is possible to develop and deploy a scalable visualization service rapidly in emergency responses where there is a need to handle a huge volume of data. Our methodologies and experience (and, indeed, our open-source software contributions) can be applied to similar situations in the future and contribute to the pandemic preparedness from an ICT perspective. Creating a data-intensive service during an emergency response, and relying on volunteering effort, is not an ideal solution, and we have taken pragmatic decisions to advance rapidly to meet the requirements of the pandemic response. In that respect, we have seen the substantial benefit of taking an agile software life-cycle approach and building on existing open source software, and our approach has now resulted in a healthy amount of open-source code as well as a working service that has been operational since August 2020. On top of our findings and experiences shared in this paper, we argue that the resulting software and system resources would be instrumental for improving our readiness for future emergency responses and can provide an infrastructural springboard to build effective visualization services in shorter time frames.

Acknowledgments

The first phase of this work (May 2020 - Jan 2021) was carried out through volunteer efforts. The second phase of this work (Feb 2021 - Jan 2022) was supported by EPSRC (EP/V054236/1). We would like to thank all volunteers in the SCRC [4] and the RAMPVIS group [5]. In particular, we would like to thank Dr. B. Bach (U. Edinburgh), Prof.

N. W. John (U. Chester), and Dr. H. C. Purchase (U. Glasgow) for their involvement in work of the generic support team. We are grateful to Dr. R. Reeve (U. Glasgow) and A. Brett (UKAEA) for their leadership in creating the SCRC data infrastructure that the RAMPVIS service depends on, and A. Lahiff and his STFC colleagues for maintaining the RAMPVIS VMs, and S. Michell (U. Glasgow) for offering valuable advice on data products.

REFERENCES

[1] L.-J. Zhang, H. Cai, and J. Zhang, *Services Computing*. Berlin, Germany: Springer, 2007.

[2] J. A. Miller, H. Zhu, and J. Zhang, "Guest editorial: Advances in web services research," *IEEE Trans. Services Computing*, vol. 10, no. 1, pp. 5–8, 2017.

[3] "Rapid assistance in modelling the pandemic: RAMP." [Online]. Available: https://epcced.github.io/ramp/

[4] "SCRC: Scottish COVID-19 response consortium." [Online]. Available: https://scottishcovidresponse.github.io/

[5] "RAMPVIS Volunteers." [Online]. Available: https://sites.google.com/view/rampvis/volunteers/

[6] M. Chen, A. Abdul-Rahman, D. Archambault, J. Dykes, A. Slingsby, P. D. Ritsos, T. Torsney-Weir, C. Turkay, B. Bach, A. Brett, H. Fang, R. Jianu, S. Khan, R. S. Laramee, P. H. Nguyen, R. Reeve, J. C. Roberts, F. Vidal, Q. Wang, J. Wood, and K. Xu, "RAMPVIS: Towards a new methodology for developing visualisation capabilities for large-scale emergency responses," arXiv:2012.04757, 2020.

[7] S. Khan, P. Nguyen, A. Abdul-Rahman, B. Bach, M. Chen, E. Freeman, and C. Turkay, "Propagating Visual Designs to Numerous Plots and Dashboards," *IEEE Trans. Visualization & Computer Graphics*, vol. 28, no. 1, pp. 86–95, 2022.

[8] M. Kuhrmann, P. Tell, R. Hebig, J. A.-C. Klunder, J. Munch, O. Linssen, D. Pfahl, M. Felderer, C. Prause, S. Macdonell, J. Nakatumba-Nabende, D. Raffo, S. Beecham, E. Tuzun, G. Lopez, N. Paez, D. Fontdevila, S. Licorish, S. Kupper, G. Ruhe, E. Knauss, O. Ozcan-Top, P. Clarke, F. H. Mc Caffery, M. Genero, A. Vizcaino, M. Piattini, M. Kalinowski, T. Conte, R. Prikladnicki, S. Krusche, A. Coskuncay, E. Scott, F. Calefato, S. Pimonova, R.-H. Pfeiffer, U. Pagh Schultz, R. Heldal, M. Fazal-Baqaie, C. Anslow, M. Nayebi, K. Schneider, S. Sauer, D. Winkler, S. Biffl, C. Bastarrica, and I. Richardson, "What makes Agile software development Agile," *IEEE Trans. Software Engineering*, pp. 1–1, 2021.

[9] F. Maurer and S. Martel, "Extreme programming. rapid development for web-based applications," *IEEE Internet Computing*, vol. 6, no. 1, pp. 86–90, 2002.

[10] A. Fruhling and G.-J. De Vreede, "Field experiences with EXtreme programming: Developing an emergency response system," *J. Manage. Inf. Syst.*, vol. 22, no. 4, p. 39–68, Apr. 2006.

[11] H. Edison, X. Wang, and K. Conboy, "Comparing methods for large-scale Agile software development: A systematic literature review," *IEEE Trans. Software Engineering*, pp. 1–1, 2021.

[12] M. Krausz, J. N. Westenberg, D. Vigo, R. T. Spence, and D. Ramsey, "Emergency response to COVID-19 in canada: Platform development and implementation for eHealth in crisis management," *JMIR Public Health Surveill*, vol. 6, no. 2, p. e18995, May 2020.

[13] T. Schinköthe, M. R. Gabri, M. Mitterer, P. Gouveia, V. Heinemann, N. Harbeck, and M. Subklewe, "A web- and app-based connected care solution for COVID-19 in- and outpatient care: Qualitative study and application development," *JMIR Public Health Surveill*, vol. 6, no. 2, p. e19033, Jun 2020.

[14] M.-J. Antonini, D. Plana, S. Srinivasan, L. Atta, A. Achanta, H. Yang, A. K. Cramer, J. Freake, M. S. Sinha, S. H. Yu, N. R. LeBoeuf, B. Linville-Engler, and P. K. Sorger, "A crisis-responsive framework for medical device development applied to the COVID-19 pandemic," *Frontiers in Digital Health*, vol. 3, p. 25, 2021.

[15] R. A. Dixit, S. Hurst, K. T. Adams, C. Boxley, K. Lysen-Hendershot, S. S. Bennett, E. Booker, and R. M. Ratwani, "Rapid development of visualization dashboards to enhance situation awareness of COVID-19 telehealth initiatives at a multihospital healthcare system," *J. American Medical Informatics Assoc.*, vol. 27, no. 9, pp. 1456–1461, 2020.

[16] "Public health scotland covid-19 daily dashboard." [Online]. Available: https://public.tableau.com/app/profile/phs.covid.19/viz/COVID-19DailyDashboard_15960160643010/Overview

[17] J. Alarte, J. Silva, and S. Tamarit, "What web template extractor should I use? A benchmarking and comparison for five template extractors," *ACM Trans. on the Web*, vol. 13, no. 2, 2019.

[18] H. Mao and L. Zhu, "Template-based framework for rapid application development platform," in *2011 Asia-Pacific Power and Energy Engineering Conference*, 2011, pp. 1–4.

[19] D.-P. Pop and A. Altar, "Designing an MVC model for rapid web application development," *Procedia Engineering*, vol. 69, pp. 1172–1179, 2014.

[20] T. Parr, "Enforcing strict model-view separation in template engines," in *13 Int. World Wide Web Conf. Proc.*, 2004, pp. 224–233.

[21] U. Zdun, "Dynamically generating web application fragments from page templates," in *Proc. 2002 ACM Symp. Applied Computing*, 2002, p. 1113–1120.

[22] H.-J. Happel and S. Seedorf, "Applications of ontologies in software engineering," in *Proc. Workshop on Sematic Web Enabled Software Engg. - SWESE*, 2006, pp. 5–9.

[23] D. Gašević, N. Kaviani, and M. Milanović, "Ontologies and software engineering," in *Handbook on Ontologies*. Springer, 2009, pp. 593–615.

[24] S. Isotani, I. I. Bittencourt, E. F. Barbosa, D. Dermeval, and R. O. A. Paiva, "Ontology driven software engineering: a review of challenges and opportunities," *IEEE Latin America Transactions*, vol. 13, no. 3, pp. 863–869, 2015.

[25] R. Witte, Y. Zhang, and J. Rilling, "Empowering software maintainers with semantic web technologies," in *Proc. of the European Semantic Web Conf.* Springer, 2007, pp. 37–52.

[26] H.-J. Happel, A. Korthaus, S. Seedorf, and P. Tomczyk, "Kontor: an ontology-enabled approach to software reuse," in *Proc. 18th Int. Conf. On Software Engg. And Knowledge Engg.*, 2006.

[27] A. P. Ambrosio, D. C. de Santos, F. N. de Lucena, and J. C. da Silva, "Software engineering documentation: an ontology-based approach," in *Proc. IEEE WebMedia & LA-Web*, 2004, pp. 38–40.

[28] Y. Zhao, J. Dong, and T. Peng, "Ontology classification for semantic-web-based software engineering," *IEEE Trans. Services Computing*, vol. 2, no. 4, pp. 303–317, 2009.

[29] Z. Sun, C. Hu, C. Li, and L. Wu, "Domain ontology construction and evaluation for the entire process of software testing," *IEEE Access*, vol. 8, pp. 205 374–205 385, 2020.

[30] L. I. Terlouw and A. Albani, "An enterprise ontology-based approach to service specification," *IEEE Trans. Services Computing*, vol. 6, no. 1, pp. 89–101, 2013.

[31] K. Ren, N. Xiao, and J. Chen, "Building quick service query list using WordNet and multiple heterogeneous ontologies toward more realistic service composition," *IEEE Trans. Services Computing*, vol. 4, no. 3, pp. 216–229, 2011.

[32] M. Michlmayr, "Managing volunteer activity in free software projects." in *Proc. of the USENIX Annual Technical Conf. - USENIX'04*, 2004, pp. 93–102.

[33] K. Brodlie, J. Brooke, M. Chen, D. Chisnall, A. Fewings, C. Hughes, N. W. John, M. W. Jones, M. Riding, and N. Roard, "Visual supercomputing – technologies, applications and challenges," *Computer Graphics Forum*, vol. 24, no. 2, pp. 217–245, 2005.

[34] S. Khan, "RAMPVIS ontology management and propagation UI." [Online]. Available: https://github.com/saifulkhan/rampvis-ontology-management-ui/

[35] S. Khan and P. H. Nguyen, "RAMPVIS api." [Online]. Available: https://github.com/ScottishCovidResponse/rampvis-api/

[36] S. Khan, P. H. Nguyen, A. Abdul-Rahman, and B. Bach, "RAMPVIS ui." [Online]. Available: https://github.com/ScottishCovidResponse/rampvis-ui/

[37] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[38] J. Webber, S. Parastatidis, and I. Robinson, *REST in Practice: Hypermedia and Systems Architecture*, 2010.

[39] J. Algermissen, "Classification of HTTP APIs." [Online]. Available: http://algermissen.io/classification_of_http_apis.html

[40] Microsoft, "N-tier architecture style." [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier/

[41] M. Flower, "Richardson Maturity Model." [Online]. Available: https://martinfowler.com/articles/richardsonMaturityModel.html
[42] A. L. Lemos, F. Daniel, and B. Benatallah, "Web service composition: A survey of techniques and tools," *ACM Computing Surveys*, vol. 48, no. 3, 2015.
[43] M. Wooldridge, "Agent-based software engineering," *IEE Proceedings-Software Engineering*, vol. 144, no. 1, pp. 26–37, 1997.
[44] N. R. Jennings, "On agent-based software engineering," *Artificial intelligence*, vol. 117, no. 2, pp. 277–296, 2000.
[45] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns, *Elements of reusable object-oriented software*. Addison-Wesley Reading, Massachusetts, 1995, vol. 99.
[46] "The situational factors that affect the software development process: Towards a comprehensive reference framework," *Information and Software Technology*, vol. 54, no. 5, pp. 433–447, 2012.
[47] O. Lassila and R. R. Swick, "Resource Description Framework (RDF) Model and Syntax Specification," Tech. Rep., 1999. [Online]. Available: https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/
[48] F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein, "OWL Web Ontology Language," Tech. Rep., 2003. [Online]. Available: https://www.w3.org/TR/2003/WD-owl-ref-20030331/
[49] N. Roy-Hubara, L. Rokach, B. Shapira, and P. Shoval, "Modeling Graph Database Schema," *IT Professional*, vol. 19, no. 6, pp. 34–43, 2017.
[50] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Computing Surveys*, vol. 40, no. 1, pp. 1–39, 2008.
[51] P. Atzeni, C. S. Jensen, G. Orsi, S. Ram, L. Tanca, and R. Torlone, "The relational model is dead, SQL is dead, and I don't feel so good myself," *SIGMOD Rec.*, vol. 42, no. 2, 2013.
[52] M. Cinque, R. D. Corte, and A. Pecchia, "Microservices Monitoring with Event Logs and Black Box Execution Tracing," *IEEE Trans. on Services Computing*, vol. 15, no. 1, pp. 294–307, Jan. 2022.

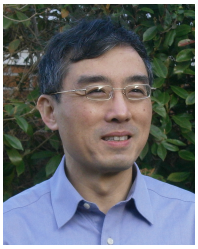
Saiful Khan received his DPhil in Engineering Science from the University of Oxford, and is currently a researcher at the University of Oxford. He worked as a Software Engineer at ABB, Oracle, International Seismological Centre and a Data Scientist at Horus Security Consultancy. He has experience of developing data processing, modeling, search, and visualization techniques in various projects for applications such as Building Information Management, radio astronomy, seismology, and security-intelligence.

Phong H. Nguyen is a Data Scientist at RedSift, and a part-time researcher at Oxford University. His research mainly focuses on the design and application of interactive visualizations to make sense of complex datasets, with a special interest in analytic provenance, logs and general temporal categorical data. He has published papers in high-impact journals including IEEE TVCG, InfoVis, VAST, CG&A and IVS. Phong holds a PhD in Visual Analytics from Middlesex University, London, UK.

Alfie Abdul-Rahman is a Lecturer in Computer Science at King's College London. She received her PhD from Swansea University in Computer Science. Before joining King's College London, she was a Research Associate at the University of Oxford e-Research Centre. She worked as a Research Engineer in HP Labs Bristol on document engineering, and then as a Software Developer in London, working on multi-format publishing. Her research interests include information visualization, visual analytics, computer graphics, human-computer interaction, and digital humanities. She developed several web applications, including Poem Viewer and ViTA.

Euan Freeman is a Lecturer in Computing Science at University of Glasgow. He has a PhD in Human-Computer Interaction from University of Glasgow. He often works with industry on future human-computer interfaces (e.g., with Logitech, UltraLeap, Huawei and Nokia). His main research interests include haptic interaction techniques, information perception, and novel methods of information representation (e.g., using non-visual modalities).

Cagatay Turkay is an Associate Professor at the Centre for Interdisciplinary Methodologies at University of Warwick. His research focuses on designing visualisations, interactions and computational methods to enable an effective combination of human and machine capabilities to facilitate data-intensive problem solving. He serves as a committee and organising member for several conferences including VIS and EuroVis. He served as a guest editor for ACM Transactions on Interactive and Intelligent Systems and IEEE Computer Graphics and Applications, and an editorial board member for Computers and Graphics and Machine Learning and Knowledge Extraction journals.



Min Chen received his PhD from University of Wales (Swansea) in 1991. He is currently a professor of scientific visualization at Oxford University and a fellow of Pembroke College. Before joining Oxford, he held research and faculty positions at Swansea University. His research interests include data science, visualization, computer graphics and human-computer interaction. His services to the research community include papers co-chair of IEEE Vis 2007 and 2008, IEEE VAST 2014 and 2015, and EG 2011; co-chair of VG 1999 and 2006, and EuroVis 2014; AEIC of IEEE TVCG and EIC of CGF; and co-director of Wales Research Institute of Visual Computing. He is a fellow of BCS, EG and LSW.