Climate Data Service Integration using the Stratus Framework

Thomas Maxwell
NASA NCCS
thomas.maxwell@nasa.gov

1. Introduction

Due to the "data tsunami", it is becoming increasingly impractical for investigators to download climate datasets to their desktop computers to perform analyses. Server-side computing, which employs high performance computing resources co-located with the data, is becoming a dominant paradigm in climate data analytics. Analytic services expose these remote computing capabilities to the end user. Many protocols, standards, technologies, and architectures have been developed to facilitate the construction and deployment of climate data analytic services. There is now a growing need to integrate this plethora of methodologies into a common framework, exposing a single unified interface to the user. This paper describes an analytic services integration paradigm designed to fill this need. We begin by presenting a exposition of the integration challenge, describing the important requirements and constraints, and then outline a solution that is under development at NASA.

2. Analytic Services

Analytic services are used to expose NCCS compute capabilities to remote users. These services enable users to perform operations on their desktop computers that are then executed remotely on NCCS high performance computing resources. A simple abstract analytic service architecture, illustrated in Figure 1, is composed of the following components:

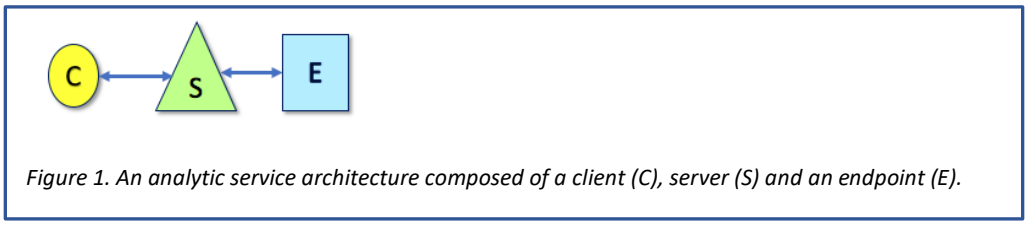


Figure 1. An analytic service architecture composed of a client (C), server (S) and an endpoint (E).

- **Client:** The use interface of the service, typically a web application, python script or Jupyter notebook that is executed by the user on their desktop computer. The client connects to the remote server using a protocol such as http.
- **Server:** An application which accepts connections from the client and manages one or more endpoints.
- **Endpoint:** The data processing operation that is being exposed to the user, e.g. a python application that reads data from a model, computes an average, and then returns the result to the user.

This service definition is labeled "abstract" because a real-world analytic service architecture would be significantly more complicated. Figure 2 illustrates a somewhat more realistic multi-layer architecture for a single analytic service as it would be structured at NCCS.

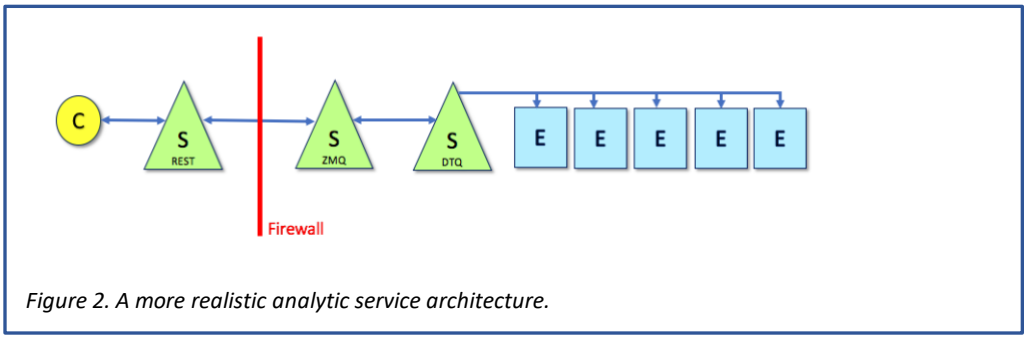


Figure 2. A more realistic analytic service architecture.

At NASA the public facing servers will always be separated from the high performance compute resources by a firewall. In this configuration the client connects to the REST server, which then makes a zeroMQ (ZMQ) connection through the firewall to a ZMQ server on the NASA cluster. In order to manage the load of many simultaneous users the ZMQ server passes the task to a Distribute Task Queue (DTQ) which manages a collection of workers distributed over the nodes of the cluster.

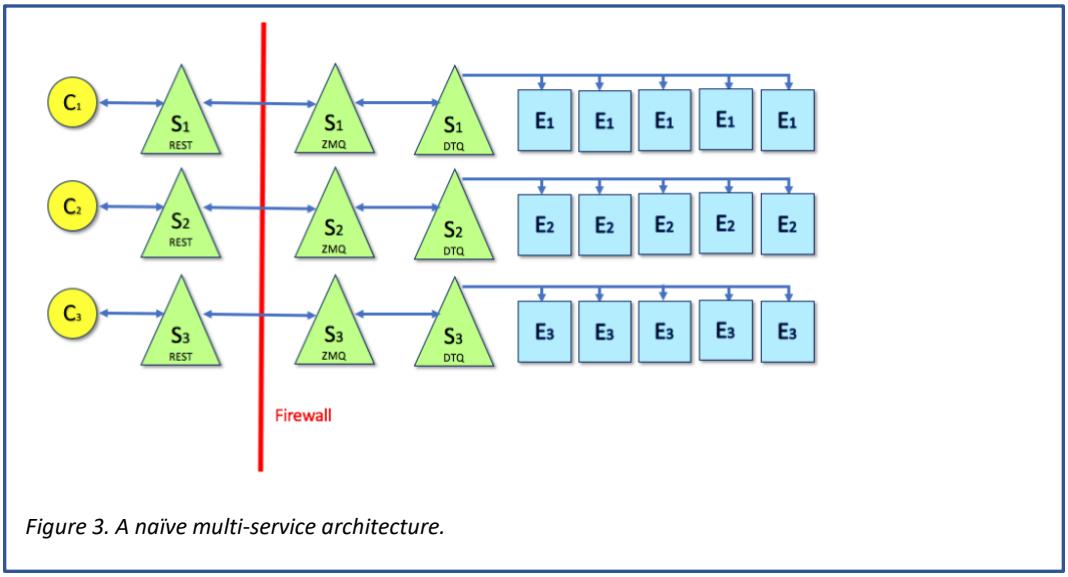


Figure 3. A naïve multi-service architecture.

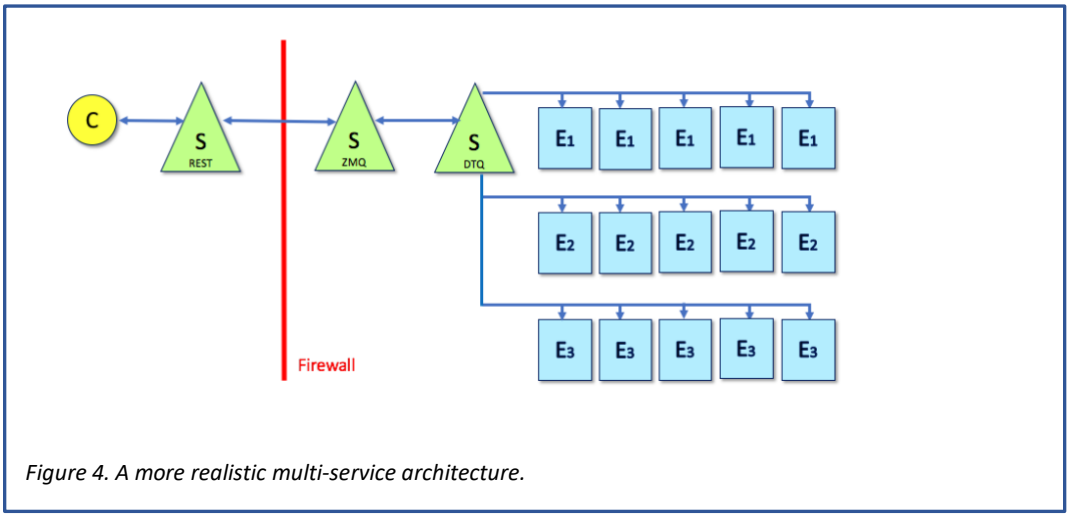


Figure 4. A more realistic multi-service architecture.

The next step is to consider the requirements of a multi-service architecture. Figure 3 illustrates a naïve architecture which simply multiplies the configuration in Figure 2. This architecture is impractical for numerous reasons, including:

- **Duplication of effort:** For each service a custom client, REST service, ZMQ service, and DTQ service must be implemented, tested, security-reviewed, documented, and deployed.
- **Security:** An architecture which requires new ports to be opened in the firewall for each new service will (justifiably) invoke strong pushback for the security team. In addition, each new service requires a new security review, which is costly and time consuming.
- **Client-side complexity:** The complex task of orchestrating and integrating the various services falls to the end user, significantly degrading the usability of the system.

A more realistic architecture is shown in Figure 4. In this case a collection of service endpoints is exposed to the user using a single client and set of server layers. This architecture requires that a common request language be established across all of the supported endpoints, which is used by the client to submit service requests to any of the endpoints. It has the advantage of naturally supporting workflows that are composed of multiple endpoints because all endpoints are managed by a single DTQ.

So far the focus has been on the framework layers. Another important consideration is the communication pattern deployed over the connections between layers. There are a number of classic patterns which are each optimal in certain contexts. Often these patterns are combined in pairs. Some of the most common patterns are:

- **Request-reply:** The is the classic REST two-way messaging pattern. The client sends a message to the server and then the server sends a single reply message back to the client. The client must wait for the reply before it can send another message.
- **Publish-subscribe:** This is a one-way data distribution pattern, in which each client subscribes to receive a certain category of messages from the server, and the server pushes messages out to the clients. The message stream arriving at each client is filtered to contain only messages of the subscribed category.
- **Push-pull:** This pattern is used to implement fan-out and fan-in topologies. In a fan-out pattern, a source node pushes (broadcasts) messages to a collection of destination nodes. This may be followed by a fan-in pattern, in which the destination nodes all push messages to a single result node.
- **Pair:** One node sends a message directly (exclusively) to one other connected node.

These and other communication patterns have multiple variations involving routing, queueing, blocking, etc. They can be combined to create a rich palate of communication possibilities for our analytics architecture. The choice of communication pattern for a given architectural layer is intimately wed to the technology used to implement that layer. For example, a REST server will only support the **Request-reply** pattern. A distributed task queue will utilize some combination of **Push-pull** and **Pair** connections. A ZMQ based layer can potentially support any of the listed patterns, but will typically be implemented using a complementary pair such as **Request-reply** and **Publish-subscribe.**

Combining analytic services with varying requirements and contexts can be expected to require a complex multi-layer architecture that integrates diverse technologies and orchestration strategies. The design of an analytic services framework must emphasize flexibility, because the optimal choice of technologies and strategies can vary over time, from site to site, and between local cluster and the cloud. The essential element for enabling flexibility is modularity, both in the endpoints and in the architectural layers. Based on these considerations we can now formulate the three primary requirements of our integration strategy:

- Modular Endpoints: Requires a common endpoint API that can be used to wrap any analytic operator of interest and expose it to the system as an instantiation of a singular analytic module interface.
- Modular Layers: Requires a technology-agnostic architectural layer specification which can be instantiated using a wide range of applicable technologies. Because these layers adhere to a common API it should be possible to assemble them like Lego blocks into a wide range of possible configurations, and easily replace one technology with another when moving between contexts, e.g. between cluster and cloud.
- A common language: Requires a common request-response language for describing workflows that can be assimilated and understood by all of the architectural layers and endpoints.

3. The Stratus Service Integration Framework

STRATUS (*Synchronization Technology Relating Analytic Transparently Unified Services*) is an integrative framework presenting a unified API and workflow orchestration for varied climate data analytic services. The Stratus approach addresses the three integration strategy requirements listed above by defining a technology agnostic framework for constructing analytic service architectures. It is composed of a set of modular, pluggable orchestration nodes each implementing a particular communication pattern on a particular technology and designed to interface with other Stratus nodes. In this way arbitrarily complex architectures can be constructed by combining orchestration nodes

like Lego blocks. For example, a Stratus instantiation of the architecture displayed in figure 2 would be composed of a Rest node connected to a ZMQ node connected to a DTQ node connected to an endpoint.

Stratus addresses the three integration strategy requirements discussed in the previous section:

- **Common language:** STRATUS defines a json-based request specification (SRS) syntax that is general enough to represent analytic operation execution requests for virtually any relevant analytic service. An example task request is shown in figure 5. The SRS requires only one parameter, called **operation**, whose value is a list of operations, each having the following properties:
    - **name:** The name of the operation, prefixed by an endpoint address (epa). The operation will be routed to a particular endpoint for execution based on the epa.
    - **result:** A string id used to identify the result of the operation.
    - **input:** A list of ids identifying the inputs to the operation. The list can include result ids from other operations, which enables the construction of workflows. The ids are used to determine operation dependencies in the workflow orchestration.

  The rest of the request (i.e. all parameters prefixed by an epa, e.g. *edas:*) is endpoint specific and not constrained by the SRS.

- **Modular Endpoints:** Stratus defines an endpoint API that can wrap virtually any relevant climate data analytic application and expose it as a Stratus endpoint. Any wrapped application can process requests from stratus and operate within workflows that are composed of multiple stratus endpoints. The primary functions of the Endpoint API are:
    - *Processing requests:* The endpoint API receives stratus task requests and executes them by calling the appropriate methods of the wrapped application's API, translating the Stratus-supplied parameters into the required application parameters.
    - *Application management:* The endpoint API responds to Stratus requests for status information on running endpoint tasks and endpoint capabilities information (e.g. available datasets and operations)
    - *Data translation:* If necessary the endpoint API translates input/output data between the Stratus internal data structure (xarray) and the endpoint application's data structure.
- **Modular Layers:** Stratus achieves modularity in architectural layers using a technology agnostic specification for both client and server. Stratus generalizes the simple client-server architecture illustrated in figure 1, implementing a more flexible structure illustrated in figure 6. In this diagram the server (S) has been relabeled as a Stratus orchestration node (N) that serves as both a server and a controller, as explained below. Each node receives requests from upstream clients and sends requests to downstream clients. Since all clients adhere to the same API, the downstream clients from one node can be the upstream clients for another node, allowing nodes implemented with different technologies to be easily interfaced. The endpoint is exposed through the Stratus Endpoint API (SEA) which features its own built-in client.
    - **Stratus Client:** Status defines a client API that is used across all technologies and orchestration strategies. This API is the glue that connects interchangeable orchestration nodes to build architectures of arbitrary complexity. The API is composed of four methods:
        - **Execute:** Submit an analytic SRS request as a new execution task.
        - **Status:** Obtain status information on a running task.
        - **Result:** Retrieve the result of a completed task.
        - **Capabilities:** Retrieve capabilities (e.g. available datasets and operations) from the available endpoints in the ASOP.
    - **Stratus Orchestration Node:** The stratus orchestration node has two functions, server and controller. The server component accepts incoming requests from connected upstream clients and passes the requests to the controller. The controller serves as a workflow manager, distributing the requests to downstream clients and managing the workflow execution. In processing an SRS requests the controller follows the following steps:

- **Workflow construction:** Build a workflow (defined as a dependency graph) of operations defined in the SRS request. In the dependency graph each operation is linked to the operations that compute its inputs.
- **Assign operations:** Using each operation's epa determine which of the downstream clients will process the operation and assign the operation to that client.
- **Consolidate requests:** All operations that are linked in the dependency graph and assigned to the same client are merged into a single task request.
- **Execute workflow:** Tasks that have no upstream dependencies are executed by passing their SRS declaration (with supporting parameters) to the execute method of the assigned client. When an task's execution is finished the status of all its operations is updated to COMPLETED. Each of the remaining tasks is executed when all its upstream dependencies reach COMPLETED status.
- **Completion:** When all tasks have completed execution a message is sent to the upstream clients signaling task completion and transferring the results (as binary data or uri references) of all operations that have no downstream dependencies.

- **Stratus Endpoints:** Stratus provide a special endpoint client which interfaces to the SEA. This enables a (wrapped) endpoint to be plugged into the architecture in place of any downstream client.

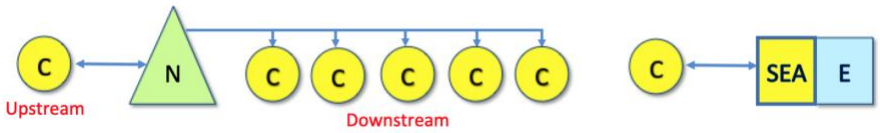


Figure 6: Stratus generalization of the simple client-server architecture in figure 1.

4. The STRATUS deployment at NASA

The Stratus framework has been partially implemented at NCCS and a limited deployment is in the initial stages of testing and security review. The deployment will occur in two stages, which are shown in figures 7 and 8.

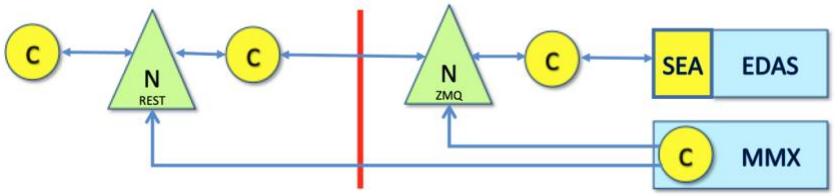


Figure 7: Architecture of phase 1 Stratus deployment .

- **Phase 1 Deployment:** In this architecture Stratus is used to expose the Earth Data Analytic Services (EDAS) framework as an analytic engine for the ESGF, and enable the MERRA-Max (MMX) application to submit data access and processing requests to EDAS. An end-user client (e.g. in a Jupyter Notebook or embedded in MMX or ESGF) connects to a stratus REST node outside of the NCCS firewall. Two protocols are supported for this connection: a json-based Stratus protocol and the xml-based wps-cwt protocol. The REST node connects through the firewall to a ZMQ node on the cluster head node. The ZMQ node connects directly to EDAS as a Stratus endpoint. MMX is controlled by an application-specific client (outside of Stratus) and uses a Stratus client to access and preprocess datasets.

- **Phase 2 Deployment:** This architecture will enable EDAS and MMX to be integrated (as Stratus endpoints) within a common request API and workflow manager. In this configuration both EDAS and MMX are under the control of Stratus. The end-user client then then submit workflows that interleave EDAS and MMX operations. Stratus can also handle the data transfer through the SEA, without requiring a Stratus client to be embedded in the MMX codebase.

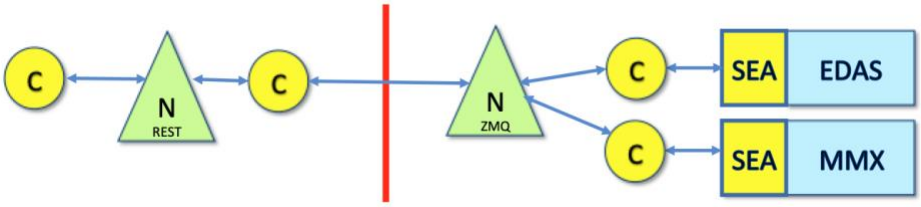


Figure 8: Architecture of phase 2 Stratus deployment .

5. Stratus Examples

This section presents examples illustrating the simplicity of deploying and integrating analytic services using Stratus, starting with several methods for deploying EDAS. The EDAS codebase includes an SEA interface so it can function as a Stratus endpoint out-of-the-box. To send analytic execution requests to EDAS we execute the Stratus ***run_client*** script (or python method), passing in the SRS request (e.g. figure 5) and configuration information specifying the stratus node that will receive the connection. To start up a stratus node we execute the Stratus ***run_node*** script or python method, passing in configuration information specifying the node type and the downstream clients. Numerous different architectures are possible:

E1: Direct Connection

In this case the client and endpoint are run together as a single application. The client would be executed with the configuration file in table 1, which specifies the class that represents the SEA within the EDAS codebase.

type = **endpoint** **module** = **edas.stratus.endpoint** **object** = **EDASEndpoint**	

Table 1: Client configuration (left) and architecture diagram (right).

E2: WPS Connection

In this case the client connects remotely to the endpoint using the wps protocol over REST. The REST Stratus node would be started on the portal machine by executing the ***run_node*** script with the configuration in Table 2 (middle). The Stratus client would be executed on the client machine using the configuration in Table 2 (left). In the node configuration the **[stratus]** section specifies the node server parameters for the upstream connection, and the **[edas]** section configures a single downstream client, named "*edas*", which in this case is a direct connection to an endpoint running on the portal machine.

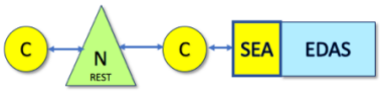
type = **rest** **host** = **portal.nccs.nasa.gov** **port** = **5000** **api** = **wps**	**[stratus]** **type** = **rest** **port** = **5000** **[edas]** **type** = **endpoint** **module** = **edas.stratus.endpoint** **object** = **EDASEndpoint**	

Table 2: REST client configuration (left), REST node configuration (middle) and architecture diagram (right).

E3: ZMQ Connection

If both the client and the server are inside of the firewall then it is more efficient to connect using ZMQ rather than REST. In this case the ZMQ Stratus node would be started on the cluster machine by executing the ***run_node*** script with the configuration in Table 3 (middle). The client would be executed using the configuration in Table 3 (left). As before, in the node configuration the [stratus] section specifies the node server parameters for the upstream connection, and the [edas] section configures a single downstream client, named "*edas*", which again is a direct connection to an endpoint.

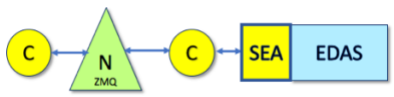
type = zmq host = cluster.nccs.nasa.gov request_port = 4556 response_port = 4557	[stratus] type = zmq request_port = 4556 response_port = 4557 [edas] type = endpoint module = edas.stratus.endpoint object = EDASEndpoint	

Table 3: REST client configuration (left), server configuration (middle) and architecture diagram (right).

E4: REST-ZMQ Connection

This is the architecture that is illustrated in figure 2 (without the DTQ). The Stratus client connects to a Stratus REST node running on the portal machine (outside of the firewall), which then uses a ZMQ connection through the firewall to reach the endpoint running on the NASA cluster. The REST node and the ZMQ node would be started using the configurations listed in table 4. The Stratus client would be executed using the same configuration as in Table 2 (with the option of using a different API, e.g. '*stratus*' rather than '*wps*'). In both of the node configurations the [stratus] section specifies the node server parameters for the upstream connection, and the [edas] section configures a single downstream client, named "*edas*".

[stratus] type = rest port = 5000 [edas] type = zmq host = cluster.nccs.nasa.gov request_port = 4556 response_port = 4557	[stratus] type = zmq request_port = 4556 response_port = 4557 [edas] type = endpoint module = edas.stratus.endpoint object = EDASEndpoint	

Table 4: REST node configuration (left), ZMQ node configuration (middle) and architecture diagram (right).

E5: EDAS-MMX Connection:

In the Stratus Phase 1 deployment plan, the Stratus client embedded in the MMX application can connect to the EDAS deployment architecture at two possible locations: it can connect to the REST node outside of the firewall or, if MMX is installed inside of the firewall, it can connect directly to the ZMQ node.. These two connections require completely different protocols and communication patterns, but switching from one to the other in Stratus is as simple as changing a couple of parameters in the client configuration.

Table 5 shows the client configurations that would enable each of these connections.

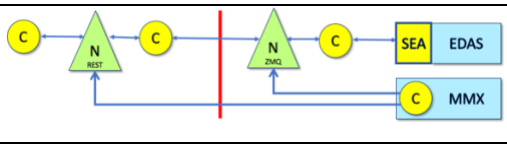
type = rest host = portal.nccs.nasa.gov port = 5000 api = stratus	type = zmq host = cluster.nccs.nasa.gov request_port = 4556 response_port = 4557	

Table 5: Client configurations: connect to REST node (left), connect to ZMQ node (middle), and architecture diagram (right).

E6: Low Level configuration

The previous examples illustrated some simple Stratus configurations, assuming default values for all unspecified parameters. It is also possible to customize the technology underlying the Stratus node by including (optional) technology-specific parameters in the configuration. An example is shown in table 6, which configures a Stratus REST node which uses the Flask framework.

```
[flask]
PROPAGATE_EXCEPTIONS = true
SQLALCHEMY_TRACK_MODIFICATIONS = true
DATABASE_URI = sqlite:////tmp/test.db

TRAP_HTTP_EXCEPTIONS = true


[stratus]
type = rest

port = 5000

[edas]

type = zmq
host = stratus.nccs.nasa.gov
request_port = 4556
response_port = 4557
```

Table 6: REST server configuration including a custom Flask configuration.

6. Conclusion

The Stratus framework enables modular construction of analytic service architectures. It provides a set of standards and APIs to facilitate the integration of disparate analytic services into unified workflows with a common interface. It enables the various technologies which compose an analytic services architecture to be reconfigured like Lego blocks, swapping technologies in and out as the architecture is redeployed in different contexts, or new technologies emerge. Developing a service architecture using Stratus has a number of advantages over the traditions approach:

- **Rapid deployment of new services:** The deployment of a new application as a service using Stratus only requires wrapping the application as a Stratus endpoint. All of the rest of the deployment architecture is constructed by piecing together existing Stratus components. Thus, standing up a new service using Stratus is much faster then the traditional method of hand-crafting a new set of architectural nodes for each new service.
- **Agile architectures:** Architectures built using Stratus can be easily adapted to new contexts and emerging technologies.
- **Rapid prototyping:** Stratus facilitates experimentation and evaluation of new technologies and configurations, as architectures can be completely restructured by rearranging existing components.
- **Robust and resilient architectures:** Stratus components undergo continual testing and optimization as they are reused in various contexts, resulting in architectures that are quite significantly more robust and resilient then the collection of one-off, custom-built components that make up a traditional architecture.
- **Bridging domains:** Because Stratus enforces a common language and supports integrated workflows with built-in data conversion, it could serve to bridge analytic domains, e.g. the GIS domain and the climate model domain.

This paper has focused on quite simple applications of the Stratus framework. The full power of this approach will become more obvious as the analytic service architectures become more complex and more technologies & applications are assimilated into the Stratus framework.

7. References

Stratus: https://github.com/nasa-nccs-cds/stratus

Stratus Endpoint: https://github.com/nasa-nccs-cds/stratus-endpoint

EDAS: https://github.com/nasa-nccs-cds/edask

MMX: https://github.com/nasa-nccs-hpda/mmx.git

WPS-CWT: https://github.com/ESGF/esgf-compute-api.git