# Microservices: An Agile Infrastructure for an Agile World

M. Jenkins[1], A. Scalise[1]

[1] PGS

## Summary

High Performance Computing is a rapidly changing and evolving field. With the introduction of Cloud Computing, it is more important than ever to make sure the software running on it can change and evolve at the same pace. Microservices are the best way to make sure software can be as cutting edge as possible. They are small, adaptable and can utilize the latest technologies and tools. They ensure that the software you run can be just as agile as the teams that make them so that they can respond quickly to the changing landscape of High Performance Computing. This paper details the advantages and need to transition High Performance Computing software to a microservice architecture and the process by which to do so.

# Microservices: An Agile Infrastructure for an Agile World

## Introduction

High Performance Computing is a rapidly evolving field. CPUs and GPUs are advancing quickly and access to the latest technology is easier and cheaper than ever with Cloud Computing. With the hardware changing so rapidly, the software needed to interface with these complex machines is often forgotten and neglected. It is easy to become complacent with software that works until there is a critical bug in a system written in a large legacy codebase that is difficult to understand. As companies make the strategic step to move to the public cloud the importance of updating legacy software stacks for this new environment and this new ecosystem of High Performance Computing becomes clear. An agile software stack is required that could change just as quickly as the compute that is used and infrastructure that could swiftly alert of a problem and isolate where it came from. It is clear that the best solution for this problem is microservices.

Migrating to this approach, however, was a demanding technical task. There are a lot of benefits that microservices provide, but as with all change it was not easy to transition to this new architecture. There is the challenge of maintaining older systems as replacements were being built, creating infrastructure that allowed the user to monitor all the applications, and with adoption of new tools. While there is still considerable work to be done, in this paper we can provide examples and illustrate how microservices can be leveraged to upgrade and replace legacy monolithic applications.

## Selecting the Right Tool for the Job

One of the most significant advantages of microservice architecture provides is its ability to use cutting-edge technology and not become dependent on any language, library, or code base. No longer is there a need to be siloed in legacy C & C++ systems; the programmer can use Python, JavaScript, and even languages that were made for Cloud Computing like Go. Instead of adding new features and maintaining the legacy C or C++ applications, the programmer can choose to break apart those applications and use tools that are better suited to the job. When the need is speed and efficiency, the choice could be C or Go. When an easy-to-use user interface is required, it is possible to use JavaScript instead of QT. Microservices allow the programmer to break apart systems into smaller pieces and create bespoke services that specifically cater to each use-case's needs. It also sets up an architecture where once an application is outdated or somehow becomes broken beyond fixing it is easy to cut out that application from its dependencies and build a whole new application to replace it. This allows the code base to be innovative and reactive instead of stale and stagnant.

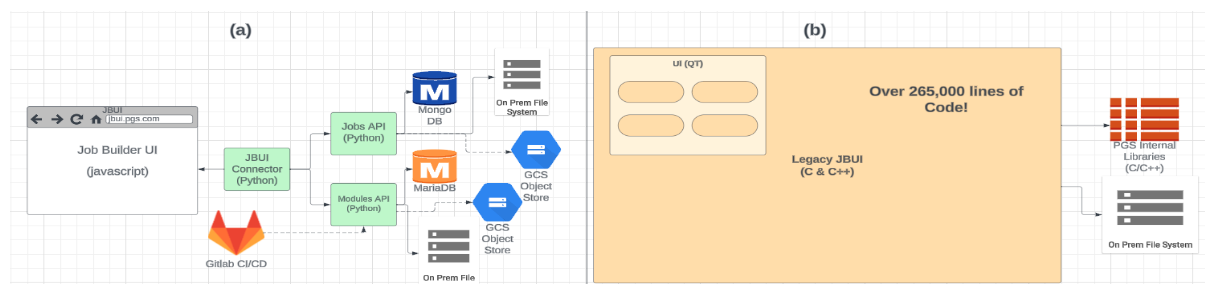


***Figure 1**: An Example of how we used microservices to replace an example legacy application. (a) The current microservice architecture that uses multiple different languages and integrates multiple tools that best suit that services need. To the right the tightly coupled legacy application coded in C and C++ that relies on legacy C libraries and the filesystem.*

In the example in Figure 1 a high-level overview of the backends for a typical Seismic Data Flow Application can be seen. To the left in Figure 1 is the new microservice approach that is composed of several different services that utilize many languages and tools. To the right is the existing implementation of the legacy application which has a large C code base and relies on the file system

and internal libraries. This example shows the agility of microservices in action. One common hurdle when moving to the cloud is the heavy reliance on on-premises filesystems. Legacy applications access the filesystems directly in many different places in the code to read job files and seismic procedure (modules) definitions. By abstracting that layer out for both jobs and modules we can create services that are catered to what they need. For example, when a highly scalable database is needed with fast look up times and little to no relations one can use a non-relational database, i.e. MongoDB. Whereas, if relations are important one can use a relational database, i.e. MariaDB. At any point in time, it is possible to switch the data source for these services from the file system to the cloud without lengthy edits or overlooked dependencies. This makes the transition to the cloud seamless.

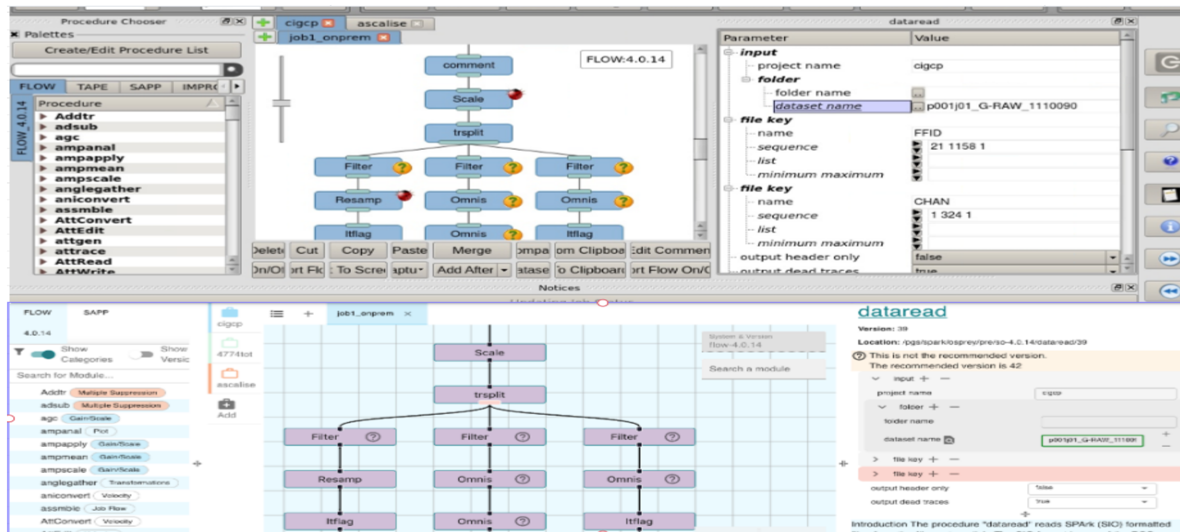


*__Figure 2__: A comparison of the old UI of a legacy Job Builder application and the new easier to use UI made possible with microservices. The top is an example of the legacy Job Builder application, and the bottom is an example of the new Job Builder application.*

In addition to a wider selection of backend tools microservices have made it possible to quickly develop modern user interfaces. Instead of relying on QT UIs that are embedded into the application, the choice can be made to use React JS and Material UI (MUI) components to create a user experience that is easy to use and highly accessible. The interfaces were then able to pass strict Web Content Accessibility Guidelines (WCAG) and, in addition, passed on some of the burden of maintenance to these mature, heavily used open-source libraries. This also had the additional benefit of embedding online geophysical support documents directly within the application reducing the amount of time going between creating a job and reviewing the documentation.

Another area that utilizes microservices is data management processes. In any cloud transition data management becomes critical. All data stored on the cloud comes with a direct cost and it is extremely important to automate the mitigation of data storage on the cloud since it is common to work with far too much data to do that process manually. Four services were created: Titan (Elastic data catalogue), Transfer, Approval, and Deletion Service, to help assist with data management. Each service focuses on one task and communicate between each other by RabbitMQ Pub Sub so that communication is asynchronous and reliable. (Dickson, 2024) This automation is necessary to ensure that costs do not balloon due to cloud storage costs.

**Maintenance Mitigation**

If your answer to solve a problem is to take one application and split it up to seven, many people will respond saying you now have 8 new problems. That reticence is not without merit. Microservices as with all things come with issues and a major issue is that if not managed correctly, they can explode in number and can be difficult to track, monitor and maintain. That is why creating a robust

infrastructure to support microservices is essential in the transition from legacy monolithic applications to light weight microservices.

The most important part of microservice architecture is transparency. It is important to be aware of issues with microservices quickly especially since microservices are interdependent on each other. A bug in one microservice can lead to service issues in other services that depend on it. It was decided to use a service, Consul, to connect and monitor all our microservices. Each service has its own health check that Consul triggers at a given interval (10sec-15sec). If the health check fails, Consul is configured to notify the necessary engineers to get the tool back in a healthy state. This allows the engineers to view the health of all the systems in one place and solve any potential issues quickly. It is also important to use a Continuous Integration Continuous Deployment (CICD) pipeline to ensure that the code that is deployed first meets a necessary set of tests or conditions so that it is difficult to accidently publish buggy or untested code.

Even with Consul there is still one major issue when transitioning to microservices: dealing with the legacy applications. Unfortunately, it is not possible to build a whole new imaging ecosystem overnight. There will always be a transition period in which both the old legacy applications must be maintained while building and maintaining new microservices. Newman (2016) suggests that engineers must identify seams (a portion of code that can be treated in isolation and worked on without impacting the rest of the codebase) in legacy applications so that the engineer can abstract the work and create service boundaries. In cases where it is possible, legacy systems have been refactored to use new microservices so that only one codebase must be maintained for any given use case. That also gives the option of connecting older services with new functionality (like access to the Cloud) and reduces the maintenance needed as newer systems are built to replace them. For example, in the example application shown in Figure 1 the old application accesses job files on the file system directly. If it were desired to add functionality for the old application to access job files on the cloud, an engineer would have to work through the old system and add the new functionality to do so as well as updating and maintaining the new microservice Jobs API to also access the cloud. Alternatively, it is possible to refactor out code that accesses jobs in the legacy application and replace it to use the new Jobs API so that one code base for this functionality needs to be maintained - the legacy system can get the new features and updates that are in the new microservice. While there will still be maintenance needed as the legacy system is in use, refactoring out the parts that can utilize the microservice architecture will allow the old systems to get new features and updates while only changing and actively maintaining one service.

**Technical Debt Considerations**

It can be hard to justify replacing a perfectly functioning system with something new. Legacy is legacy despite it being often large, ugly, and difficult to understand, it works. However, that line of thinking can lead to disastrous consequences. Martin (2018) states “If you give me a program that works perfectly but is impossible to change, then it won’t work when the requirements change, and I won’t be able to make it work. Therefore [,] the program will become useless.”. In a system that consists of legacy applications the challenge of maintaining code grows as the system ages. As companies transition from on-premises to the cloud it is clear that it is not feasible to add on support features to older applications, many of which still had extensive lists of bugs and features already. It is an ideal opportunity to replace legacy code with infrastructure that can grow with a new journey to the cloud.

Microservices can revolutionize code bases. They allow engineers to work in modern languages that are easier to maintain and can be tailored for the application they are developing for. It allows teams to leverage the skillsets of more experienced developers and for innovation and modernization that can be used and maintained for decades to come. It also enables teams to build the competency of junior engineers by giving them focused problems and smaller applications to solve, instead of handing them small problems or features to add to a large and confusing legacy code base. It can dramatically change the scope of the codebase needed to be maintained and monitored. In the legacy

application shown in Figure 1 there are over 265,000 lines of code, whereas the microservice replacement for it only has 20,998 lines of code across each of the composite services. Most importantly those services can be reused in other applications so there is not duplicate code implementations that would have to be changed in the event of a bug or feature.

A core principle in Agile is the concept of a two-pizza team e.g., a team small enough that only two pizzas would be needed to feed them. This can be hard to achieve when working in large monolithic applications that encompass many use cases in one large monolithic application. Multiple developers and developer teams would be needed to implement features and maintain the large outdated application. Newman (2016) sums up the experiment of Alan MacCormack, John Rusnak, and Carliss Baldwin in their paper Exploring the Duality Between Product and Organizational Architectures (Harvard Business School) "...[they] found the more loosely coupled organizations actually created more modular, less coupled systems, whereas the more tightly focused organization's software was less modularized.". Their experiment showed that an organization's software reflects heavily on the teams and organizations building them. If the desire is to work in small and focused teams as Agile demands, the software must be equally small and modularized. microservices enables this and allows teams to take ownership of the codebase. Teams can be smaller and focused enabling them to respond to change quickly.

**Conclusions**

Now that the world of HPC is rapidly evolving and introducing new ways to compute with Cloud Computing it is imperative that the software architecture it relies on keeps up with the pace of change. It is clear that one of the best ways to create robust, easy to adapt, modern applications is to harness the power of microservices. The second tenant of the Agile Manifesto (a rule book and bible for creating and structuring Agile Development) is “Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage” (Beck, K. *et al*, 2013) - if developing teams are going to be agile and innovative, they need a code base that empowers them to embrace change.

The journey to a microservice architecture that is described in this paper is far from over, but it is already easy to see the difference it has made in the code base. The code base is now made up of small single purpose services that are easier to maintain and easier to track down and solve problems when they arise. Microservices have helped in the journey to the cloud allowing teams to work in a hybrid manner as the transition from on-premises compute to the public cloud is undertaken. Microservices allowed engineers to work in many different languages and toolsets making it easy to create a bespoke environment for unique challenges and needs.

**References**

Dickson, J, Kontogiannopoulou, E. [2024] Efficient managing of seismic data for imaging in the public cloud. Extended abstract submitted to the EAGE annual meeting.
Beck, K.L., Beedle, M.A., Bennekum, A.V., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S.J., Schwaber, K., Sutherland, J., & Thomas, D.A. [2013]. Manifesto for Agile Software Development.
Martin, R. C. [2018] Clean architecture: A craftsman’s guide to software structure and Design. Prentice Hall.
Newman, S. [2016] Building microservices: Designing fine-grained systems. O’Reilly.