BEST PRACTICES

Best Practices that span the entire lifecycle of scientific software: Design, Development, Deployment and Discovery phases.

DISCOVERY PHASE

1. We write with our users two types of documents: use case specification and software requirements specification. Use case documents describe deployment and execution scenarios alongside the abstract workflow users want to implement. Software requirement documents describe in progressive detail the software system we are going to co-design and then develop to satisfy one or more use case documents. Both documents use templates tailored to serve cross-domain research and are constantly updated in collaboration with the users. This accommodates the progressive process of mutual understanding between developers and domain-scientists and enables rapid prototyping of software solutions.
2. Initially, we simplify use cases, requirements, and capabilities to deliver simple, baseline results in the very first phases of the project. This approach helps to reduce feature creep and gives the opportunity to the user to develop an immediate understanding of the relation between design properties and scientific results. During the project, we progressively and carefully increase complexity, at a pace dictated by the users.
3. The time spent eliciting requirements and that taken to deliver solutions needs to be minimized. We manage this tension by engaging users across software development activities and by adopting iterative processes with relatively short cycles.
4. Two types of meetings are arranged for coordinating all management and technical stuff. Management meetings have a bi-weekly cadence while we found that a weekly schedule is better for technical discussions.
5. Communication is key when developing and improving CDI applications. It leads to getting the right requirements from the user.

6. Set up regular in-person meetings or online video calls to keep communication with the user personal.
7. Empathy, if you care for your users they will care for your product.

DESIGN PHASE

1. User interface design- high performance computing (HPC) has largely languished with traditional command line interfaces that are not familiar with an increasing majority of individual.
2. The primary client interface for HPC (SSH) lacks any ability to take advantage of modern authentication like [Oauth2].
3. Make the design integrable so that researchers can keep working in their chosen computational environment and can receive additional features instead of having to switch to a different software.
4. Provide a user-centric view to support research more effectively by considering usability, scalability, and interoperability.
5. Usable design and execution efficiency are simultaneously achieved when code design includes a specific kind of separation of concerns. Separation of concerns generally refers to breaking an application into sections that address specific concerns (modular programming). The concerns we propose separating are the primary computation, the execution schedule, and the storage mapping. The primary computation is the algorithm or mathematics being performed; the order the expressions composing the primary computation are executed is the execution schedule, and the storage mapping is how the values required and written by the computation are stored in memory.
6. Developing Tools with shallower learning curves will also make it easier for us to educate our PIs and their teams in the use of the tools.
7. Being open to the idea of working with other organizations to expand the scope beyond your organization and create a broader repository that can be used as a starting point for others who need to carry out similar work.
8. As the hardware platforms on which the CDI applications typically run have a short lifespan of 4-5 years, it is prudent to “future-proof” the applications so that they continue to be usable and scalable on the future hardware platforms and hence enjoy a long-life.
9. At the software design stage, one can consider adopting the software engineering principles such as, the Open- Close Principle (OCP), as per which, the software is designed such that it is

closed for modification but open for extension. For building, deploying and distributing self-contained applications that can run on different but comparable hardware architectures, one can consider containerization. For scalability, the applications can be designed and developed to work in distributed computing environments.

10. Designing a "future-proof" software can be difficult. As one of our colleagues said: “one of the issues is not thinking exponentially”.
11. Minimize the steps it takes to complete a task.
12. Keep the interface clean. Having a busy or noisy interface can cause frustration when trying to learn a new application.
13. Every activity is shared and agreed upon with the user, openly documented and managed via GitHub tickets and wiki pages. This offers a user- and developer-friendly environment, fully integrated with code versioning and project management activities.

DEVELOPMENT PHASE

1. Successful approaches are characterized by being technology agnostic, using APIs and standard web technologies or delivering a complete solution for serving a community efficiently.
2. Scientific applications often use embedded domain-specific languages (eDSLS) that achieve separation.
3. Disseminate the information that was gained through the projects. It gives the staff an opportunity to publish findings and results that are not necessarily a good fit for journal publications or conference proceedings. For example, these reports may go into the low-level details illustrating how the code was modified to achieve higher performance or improved scalability. These technical reports can capture valuable information that might otherwise have been lost.
4. Having a strong connection with your users will provide you with some great feedback.
5. Adopting cloud-based development environments, such as Eclipse Che and Amazon Cloud9, for collaborative software development, documentation, debugging and testing. Tools for continuous integration, such as Travis CI, also become cloud-based such that testing is triggered on cloud servers when commits are pushed to the source code repository. Those tools significantly improve development productivity and reduce the amount of human involvement for routine setup and maintenance tasks. Due to the collaborative nature, software tools and applications for scientific research should consider cloud-based development, documentation,

and testing. There would be some benefits for it but also some disadvantages. The benefits are tremendous, such as:

1. Enabling real-time development on the same project, reproducing results, and debugging of a problem from the same session;
2. Enabling access to a development environment using a web browser from anywhere;
3. Improving the productivity by using and sharing of complex scientific software or hardware stack needed to develop tools and applications;
4. Facilitating training of students and new developers;
5. Enabling quick access to special hardware (such as latest GPUs) without paying the cost of buying the hardware;
6. Enabling a seamless connection with other cloud-based services such as source code, hosting, continuous integration, and deployment of tools as cloud-based service.

6. Use libraries that have been developed by others where possible. This leads to efficiencies in time to solution.
7. Try to write code that is reusable.
8. Documentation. If a code is not well documented, it becomes useless as time goes on. If one gets into the habit of adding comments directly in the code, this may be very helpful to other developers.
9. Accessibility Tests: Know your users. Find out if your users have any disabilities that could impair their use of your application.
10. Usability Test: making sure that the application is easy to use and easy to adapt.
11. Improve the application without having to change the look and feel of it. Nothing is more frustrating than having to relearn an application after a major update.
12. Each system has a set of explicitly defined: (1) entities;(2) functionalities that operate on these entities; and (3) states, events and errors for each entity. These systems become building blocks because they can be integrated with minimal code writing both among themselves or with systems independently developed by third-party developers. While this approach tends to be ‘design-heavy’ and an unstructured and rapid development approach
13. Basing our development and user support on GitHub. We use tailored branch and release models, pull requests, unit and integration tests, continuous integration, and style guide enforcement. We use documentation generators for API and internals, read the docs and GitHub wiki sites. User support is managed and performed via GitHub tickets.

14. Enforcing a taxonomy for labeling all the tickets and pull requests (mainly for the bug fixing), allowing for a statistical understanding of our development process and user support effort. For example, by mining our labels we can understand ticket distribution across core developers, third-party developers and users per year and software system; ticket response and lifetime; the correlation between bugs and portion of the code or specific topics; the correlation between topics feature requests or documentation request.
15. The uptake of containerization approaches such as Docker or Singularity allows for providing the full environment for a computational method addressing the portability between different HW architecture.
16. Containerization addresses reproducibility of science and preservation of software environments. The long-term aspect of preservation of software for over 15-20 years is probably not well addressed via containerization - immanent in dependencies on container versions, operating systems, and existing research infrastructures - but delivers an intermediate solution.

DEPLOYMENT PHASE

1. There is currently no common APIs and exchangeable formats to facilitate the interoperability of the tools (like program Analysis Tools). These constraints make it painfully difficult for CDI developers to leverage available tools to improve their applications. Deploying these tools have many practical obstacles. Installing and configuring each tool on a variety of machines is tedious and error prone. In addition, some tools require privileged access, which is hard to obtain on a managed cluster. Finally, combining results from multiple tools is even more challenging.
2. Define a common reporting format to facilitate communication among tools. Similarly, tools should expose various APIs so users (including both developers and other tools) can freely leverage tools' functionality and query their internal status.
3. The cloud-based access to tools is extremely convenient for us to quickly get results by using only a browser. However, there are still many open challenges, including the cost of running cloud-based services, the security of the servers, the privacy of clients, and the complexity of the application building/execution process.
4. Work closely with infrastructure support teams, scientific support teams, system administrators and development communities of the third-party software tools we use in our middleware. We develop trust and long-term relationships by contributing testing code for the issues we find, replicators and extensive testing documentation.

5. Further, we adopt a strictly user-driven support policy, deprioritizing every consideration that is not directly related to enabling users to perform science on their target resources. We engage with third-party tools developers by sharing our use cases and describing in detail our implementation approach. Also, in this case, we contribute testing effort and, when possible, contribute code adhering to open source community best practices.
6. Strive to shorten the time that a user identifies issues and how long it takes to fix it.