Best Practices for Scalable Scientific Software Development: A RADICAL Approach

Overview: The RADICAL Lab [1] develops RADICAL-Cybertools (RCT), a set of software systems that support the development of tools and the execution of applications on HPC platforms. RCT are building blocks, designed to work as stand-alone systems, integrated among themselves or integrated with third-party systems. RCT has three main components: RADICAL-SAGA (RS), RADICAL-Pilot (RP) and RADICAL-Ensemble Toolkit (EnTK). RS provides an access-layer and mechanisms for job schedulers, file transfer and resource provisioning services; RP is a pilot system for the distributed execution of heterogeneous bags of task on one or more HPC resources; and EnTK is a workflow engine that exposes a pipeline-based API for the development of domain-specific workflow applications. Consistently, our group supports both system and application development, requiring a coordinated set of best practices that can serve the distinct requirements of both types of development activities. This paper provides an overview of those best practices that span the entire life-cycle of scientific software: development, deployment and discovery phases.

Best Practice: Software Development

We ground our software development activity on a design methodology we called the 'building blocks approach' [2]. Built upon component based software engineering (CBSE) and service oriented architecture (SOA) concepts, our approach helps to design software systems that are self-sufficient, interoperabile, composabile, and extensibile. 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 becomes 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 infeasible, it serves well our core business: developing production-grade middleware to enable the execution of applications for diverse scientific domains [3] on high performance computing infrastructures.

We enforce an uniform approach across all our tools, basing our development and user support on GitHub. We use tailored branch [1] and release [2] models, pull requests, unit and integration tests, continuous integration, and style guide enforcement. We use documentation generators for API and internals, readthedocs and GitHub wiki sites. User support is managed and performed via GitHub tickets.

We enforce a taxonomy for labeling all our tickets and pull requests, 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 life time; correlation between bugs and portion of the code or specific topics; correlation between topics feature requests or documentation request. We use this insight to drive and steer our development effort, prioritizing feature requests and, accordingly, development of new code and rewriting of existing code.

Best Practise: User Driven Co-Design and engagement and delivering science

In an academic environment we often experience a tension between the need to minimize the time spent eliciting requirements and that taken to deliver solutions that can enable actual science. We

manage this tension by engaging users across software development activities and by adopting iterative processes with relatively short cycles. We create one or more projects for each collaboration and, within each project, 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.

We deliver science as early as possible. 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 the pace dictated by the users. We design middleware that abstracts details not directly related to the development of scientific code, enabling the involvement of domain-scientists in the prototyping of user-facing interfaces and high-level functionalities. In turn, this creates a clear separation between the code the user 'feels' that has to co-own and the runtime system code on which we have to maintain exclusive ownership.

We coordinate these activities with two types of meetings: management and technical. Management meetings have a bi-weekly cadence while we found that a weekly schedule is better for technical discussions. 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.

Best Practise: Engagement with Resource Providers and 3rd party software

We 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. Further, we adopt a strictly user-driven support policy, deprioritizing every consideration that is not directly related to enable 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.

Related References:

[1] RADICAL Laboratory: http://radical.rutgers.edu
[2] Middleware Building Blocks for Workflow Systems
https://arxiv.org/abs/1903.10057
[3] RADICAL-Cybertools: Middleware Building Blocks for Scalable Science
https://arxiv.org/abs/1904.03085