ExaWorks Software Development Kit: A Robust and Scalable Collection of Interoperable Workflows Technologies

Matteo Turilli [1,2,*], **Mihael Hategan-Marandiuc** [3,4], **Mikhail Titov** [1], **Ketan Maheshwari** [5], **Aymen Alsaadi** [2], **Andre Merzky** [6], **Ramon Arambula** [7], **Mikhail Zakharchanka** [7], **Matt Cowan** [1], **Justin M. Wozniak** [4], **Andreas Wilke** [3,4], **Ozgur Ozan Kilic** [1], **Kyle Chard** [3,4], **Rafael Ferreira da Silva** [5], **Shantenu Jha** [1,2] and **Daniel Laney** [7]

[1]*Brookhaven National Laboratory, Upton, NY, USA*
[2]*Rutgers University, New Brunswick, NJ, USA*
[3]*University of Chicago, Chicago, IL, USA*
[4]*Argonne National Laboratory, Lemont, IL, USA*
[5]*Oak Ridge National Laboratory, Oak Ridge, TN, USA*
[6]*Incomputable LLC, Highland Park, NJ, USA*
[7]*Lawrence Livermore National Laboratory, Livermore, CA, USA*

Correspondence*:
Matteo Turilli
mturilli@bnl.gov

ABSTRACT

Scientific discovery increasingly requires executing heterogeneous scientific workflows on high-performance computing (HPC) platforms. Heterogeneous workflows contain different types of tasks (e.g., simulation, analysis, and learning) that need to be mapped, scheduled, and launched on different computing. That requires a software stack that enables users to code their workflows and automate resource management and workflow execution. Currently, there are many workflow technologies with diverse levels of robustness and capabilities, and users face difficult choices of software that can effectively and efficiently support their use cases on HPC machines, especially when considering the latest exascale platforms. We contributed to addressing this issue by developing the ExaWorks Software Development Kit (SDK). The SDK is a curated collection of workflow technologies engineered following current best practices and specifically designed to work on HPC platforms. We present our experience with (1) curating those technologies, (2) integrating them to provide users with new capabilities, (3) developing a continuous integration platform to test the SDK on DOE HPC platforms, (4) designing a dashboard to publish the results of those tests, and (5) devising an innovative documentation platform to help users to use those technologies. Our experience details the requirements and the best practices needed to curate workflow technologies, and it also serves as a blueprint for the capabilities and services that DOE will have to offer to support a variety of scientific heterogeneous workflows on the newly available exascale HPC platforms.

Keywords: Scientific Workflow, Software Development Kit, High-Performance Computing, Exascale, Testing, Documentation

1 INTRODUCTION

22 Workflow systems executed at unprecedented scale are increasingly necessary to enable scientific
23 discovery (Badia Sala et al., 2017). Contemporary workflow applications benefit from new AI/ML
24 algorithmic approaches to traditional problems but also bring new and challenging computing requirements
25 to the fore (Ferreira da Silva et al., 2021). Exascale High-performance computing (HPC) platforms can
26 satisfy the growing need for scale but at the cost of requiring middleware with increased complexity and
27 multiple dimensions of heterogeneity. A renewed impulse to develop workflow applications for HPC
28 platforms meets a traditionally fragmented software ecosystem, creating several issues for the users and
29 middleware developers.

30 Domain scientists who code HPC workflow applications face four main challenges: (1) choosing a
31 workflow system that satisfies the requirements of their use cases and their target HPC platform(s);
32 (2) learning to use the chosen workflow system to code and execute their applications; (3) deploying the
33 workflow system and its middleware stack on the target HPC platform, testing that they work as intended;
34 (4) maintaining their workflow application over time while ensuring that the chosen workflow system
35 remains functional through the routine software updates of their target HPC platform.

36 In a fragmented ecosystem with many workflow systems and workflow-related technologies specifically
37 designed to support scientific applications, domain scientists must review multiple software solutions
38 with similar capabilities but diverse maturity levels, robustness, reliability, documentation, and support.
39 Furthermore, users must learn the basic functionalities of several systems to evaluate, deploy, and test
40 them on one or more HPC platforms. Together, that requires a significant effort involving significant
41 human and time resources. Further, even after choosing a viable software stack, the need for portability
42 due to the different capabilities of the HPC platforms and the availability of multiple allocations and
43 recurrently updated machines leads to a constant process of deploying, testing and fixing both the workflow
44 applications and the middleware that support their execution.

45 Given the needed resources, scientists often meet those challenges by relying on 'word of mouth,'
46 repeating the choices made by colleagues even if sub-optimal. Alternatively, users resort to coding new
47 single-point solutions, furthering the balkanization of the existing software ecosystem. Ultimately, both
48 options make it difficult, if not impossible, to express the potential of HPC scientific workflow applications
49 to support innovation and discovery. We propose that overcoming that limitation requires a novel approach
50 to make available a selected set of tools that can provably support the execution of scientific workflow
51 applications on the largest HPC platforms currently available to the scientific community. Further, we need
52 a way to grow the existing software ecosystem, avoiding effort duplication while improving scalability,
53 portability, usability, and reliability and lowering the learning curve and the maintenance effort.

54 This paper describes the lessons learned while delivering ExaWorks (Al-Saadi et al., 2021), a project
55 designed to implement the approach described above. Specifically, this paper focuses on the ExaWorks
56 Software Development Kit (SDK), an effort to select a complementary set of workflow technologies,
57 make their integration possible while maintaining their individual capabilities, testing them via continuous
58 integration (CI), making the result of those tests publicly available, and documenting them in a way that
59 helps to lower the access barriers when used on HPC platforms. This work serves not only as a blueprint
60 for realizing the full potential of modern scientific workflow applications but also as a critical overview of
61 the pros and cons of that approach, as experienced during a three-year-long project.

62 The rest of the paper is organized as follows. In §2, we briefly review similar efforts, outlining similarities
63 and relevant differences. §3 describes each software tool currently included in the ExaWorks SDK. In §4,

64 we detail one of the main technical achievements of the ExaWorks project: integrating the SDK software
65 components to provide users with a variety of new capabilities while reducing to a minimum the new code.
66 §5 offers an overview of exemplar success stories about using SDK in real-world scientific research. §6
67 describes the other two main deliverables of the ExaWork project: a testing infrastructure based on a CI
68 infrastructure for Department of Energy (DOE) HPC platforms and a monitoring infrastructure based on a
69 public dashboard, collecting the results of the CI runs. In §7, we illustrate the system we built to document
70 the SDK based on containerized executable tutorials. Finally, §8 summarizes SDK's characteristics, the
71 lessons we learned while designing and developing ExaWorks SDK, and the challenges that still lie ahead.

2   RELATED WORK

72 In this section, we briefly discuss some of the projects undertaken in the community and how the
73 SDK follows in the steps of those experiences. The Workflows Community Initiative (WCI) (WCI,
74 2024) represents a pioneering collaboration to propel advancements in workflow systems and associated
75 technologies. This initiative unites diverse stakeholders, including researchers, developers, and industry
76 practitioners, to enhance dialogue and cooperation on pivotal aspects of workflow management. These
77 aspects encompass the design, execution, monitoring, optimization, and interoperability of workflow
78 systems. A cornerstone of the initiative is the organization of the Workflows Community Summits (da Silva
79 et al., 2023). These summits are international workshops that serve as a dynamic platform for attendees to
80 exchange insights, discuss the latest trends and challenges, and forge new partnerships. ExaWorks directly
81 collaborated with and participated in WCI's activities, eliciting the requirements for its SDK.

82   SDK addresses several of the community's recommendations for the technical roadmap defined during
83 the 2021 and 2022 summits (Ferreira da Silva et al., 2021; da Silva et al., 2023). SDK provides software
84 products and technical insight for the interoperability of workflow systems, delivering integrations among
85 all its workflow technologies (see §4). Indirectly, SDK software integrations also contribute to furthering
86 the understanding of the roles played by standardization, placing the accent on integrating a diversity of
87 programming models and interfaces instead of searching for a single encompassing solution. For example,
88 the DOE Integration Research Infrastructure effort to standardize interoperable workflows is considered
89 a potential venue. SDK packaging and testing infrastructure (see §6) contributes tangible capabilities to
90 improve the support of workflow technologies on HPC platforms, and SDK documentation centered on
91 containerized tutorials (see §7) contributes to training and education. While SDK does not contribute
92 directly to the FAIR, AI, and Exascale challenges, its workflow technologies (see §3) have a roadmap to
93 develop capabilities that will contribute to addressing those challenges.

94   Separately to the community, "standards" for workflow languages have been developed, such as the
95 Common Workflow Language (Amstutz et al., 2016) and Workflow Definition Language. Additionally, a
96 few standard formats have been tried and tested, including JSON and YAML (Blin et al., 2003; DeLine,
97 2021; Ristov et al., 2021). In the previous decades, large projects such as SHIWA (Korkhov et al., 2012)
98 undertook the task of workflow interoperability by developing an interoperable representation of a workflow
99 language that multiple workflow managers could enact. ExaWorks SDK takes a different approach in
100 which tools with diverse user-facing application programming interfaces (API) and approaches to workflow
101 specification are included in the SDK and, when needed, integrated with other tools that provide runtime
102 capabilities. In that way, users can 'compose' an end-to-end software stack that best fits their workflow
103 application requirements.

104   Software development activities as part of the ECP project resulted in several scientific software tools
105 that were, in turn, used by other projects that were built on top of them. For instance, AMReX (Zhang et al.,
106 2021) mesh refinement modeling forms the basis for several other tools developed on top of it, such as
107 FerroX (Kumar et al., 2021), ExaAM, and many more, forming a collection of tools for science. xSDK (The
108 xSDK Team, 2017) is a similar effort to ours. ExaWorks SDK is designed to take inspiration from those
109 experiences. Still, it differs from them because integration is considered at the level of middleware
110 components instead of focusing on the (math) libraries level. Therefore, the term 'SDK' assumes a more
111 general meaning than the one usually associated with a set of libraries that enables the writing of a new
112 application.

113   Considering ECP, ExaWorks SDK is more akin to the DOE-sponsored Extreme-scale Scientific Software
114 Stack (E4S) (Heroux, 2023), a community effort that provides open-source software packages for
115 developing, deploying, and running scientific applications on HPC and AI platforms. E4S provides
116 from-source builds, containers, and preinstalled versions of a broad collection of HPC and AI software
117 packages (The E4S Project, 2024). Compared to EC4, ExaWorks SDK takes a loosely coupled approach
118 where tools maintain a high degree of independence and are not organized into a named distribution.
119 For example, the SDK does not mandate a packaging format for its components, as E4S does. Further,
120 ExaWorks SDK focuses on workflow technologies, not on those utilized by the tasks of those workflows.

121   Not all workflow technologies and their middleware implementations have been designed to foster
122 integration with other tools. For example, Python packages such as Dask (Rocklin et al., 2015) come
123 prepackaged with some rudimentary API to interface with the resource manager. However, they are limited
124 in functionality and tightly integrated with Dask. For that reason, SDK is a curated set of workflow
125 technologies, and, as described in the next section, part of the effort goes into evaluating whether a
126 prospective component can be integrated with other components without significant engineering effort.

3 CORE COMPONENTS

127 Exaworks SDK seeding technologies were chosen based on the requirements elicited by engaging with
128 the DOE Exascale Computing Project (ECP) workflow communities. We created a survey to identify
129 existing workflow systems efforts, both ECP-related and within the broader DOE software ecosystem.
130 That survey helped us understand the challenges, needs, and possible collaborative opportunities between
131 workflow systems and the ExaWorks project. We took a broad view of workflows, including automated
132 orchestration of complex tasks on HPC systems (e.g., DAG-based and job packing), coupling simulations
133 at different scales, adaptive/dynamic machine learning applications, and other efforts in which a variety
134 of possibly related tasks have to be executed at scale on HPC platforms. For example, after eliciting a
135 brief description of an exemplar workflow, we asked about internal/external workflow coordination, task
136 homogeneity/heterogeneity, and details about the adopted workflow tools.

137   Alongside the outcome of more than twelve community meetings, the results of our survey highlighted
138 the state of the art of scientific workflow in the ECP community. To summarize, many teams are creating
139 infrastructures to couple multiple applications, manage jobs—sometimes dynamically—and orchestrate
140 compute/analysis tasks within a single workflow and manage data staging within and outside the HPC
141 platforms. Overall, there is an evident duplication of effort in developing and maintaining infrastructures
142 with similar capabilities. Further, customized workflow tools incur significant costs to port, maintain, and
143 scale bespoke solutions that serve single-use cases on specific platforms and resources. These tools do not
144 always interface with facilities smoothly and are complex and/or costly to port across facilities. Finally, the

145 lack of proper software engineering methodologies leads to repeated failures, difficulty in debugging, and
146 expensive fixes. Overall, there was agreement in the community that costs could be minimized, and quality
147 could be improved by creating a reliable, scalable, portable software development kit (SDK) for workflows.

148   Based on the requirements summarized above, the ExaWorks SDK collects software components that
149 enable the execution of scientific workflows on HPC platforms. We consider a reference stack that allows
150 the development of scientific workflow applications, resolving the task dependencies of that application,
151 acquiring resources for executing those tasks on a target HPC platform, and then managing the execution of
152 the tasks on those resources. The SDK includes components that deliver a well-defined class of capabilities
153 of that reference stack with different user-facing interfaces and runtime capabilities. Consistently, the SDK
154 is designed to grow its components, including software systems that are open source (i.e., released under a
155 permissive or copyleft license) and developed according to software engineering best practices, such as
156 test-driven development, release early and often, and version control.

157   SDK faces a fragmented landscape of workflow technologies, with diverse systems that offer overlapping
158 capabilities. To help reduce duplication, SDK collects systems that can be integrated but have distinguishing
159 capabilities, expose diverse APIs, and support different programming models. By integrating these
160 systems, SDK reduces the need to duplicate existing capabilities within each tool. Nonetheless, each tool's
161 distinguishing capabilities can be used to support use cases that require a specific API, programming model,
162 scaling performance, or programming language. Ultimately, SDK does not commit to a vision in which a
163 single technology can solve all the requirements or all the use cases. Instead, we commit to fostering an
164 ecosystem of integrable and complementary tools that can be used independently or combined, depending
165 on the specific requirements of each use case.

166   Currently, the SDK contains seven software components: Flux (Ahn et al., 2014), MaestroWF (Di Natale
167 et al., 2019), Parsl (Babuji et al., 2019), PSI/J (Hategan-Marandiuc et al., 2023), RADICAL
168 Cybertools (Balasubramanian et al., 2016; Merzky et al., 2021), SmartSim (Partee et al., 2022) and
169 Swift (Wozniak et al., 2013). As shown in Fig 1, each system delivers capabilities end-to-end or at a
170 specific level of our reference stack. That way, we offer users alternative tools across or along the reference
171 stack, depending on their specific requirements. For example, Parsl, RADICAL Cybertools, SmartSim,
172 Swift, and, to a certain extent, MaestroWF all offer end-to-end workflow capabilities but with different
173 programming models, application programming interfaces (API), support for HPC platforms, and scientific
174 domain. Flux and PSI/J offer capabilities focused on resource and execution management, enabling the
175 execution of user-defined workflows on various HPC platforms.

176   Further, each component is designed with sub-components, exposing well-defined interfaces. That allows
177 us to promote integration among components, obtain new capabilities, and avoid lock-in into solutions
178 maintained by a single team or designed to support a specific class of scientific problems and platforms.
179 Here, we briefly describe each tool, showing how their capabilities fit the reference stack. In the next
180 section, we detail the integration among some of these components.

181   **Flux** (Ahn et al., 2014, 2020) is a next-generation resource management and scheduling framework
182 project under active development at LLNL. It is composed of a modular set of projects, tools, and libraries
183 that can provide both system- and user-level resource managers and schedulers under one common software
184 framework: system administrators and end users alike can create their instances of Flux on a set of
185 HPC resources using the same commands and APIs to manage them according to their requirements.
186 Furthermore, the owners of a Flux instance (e.g., a specific user who gets a set of compute nodes allocated)
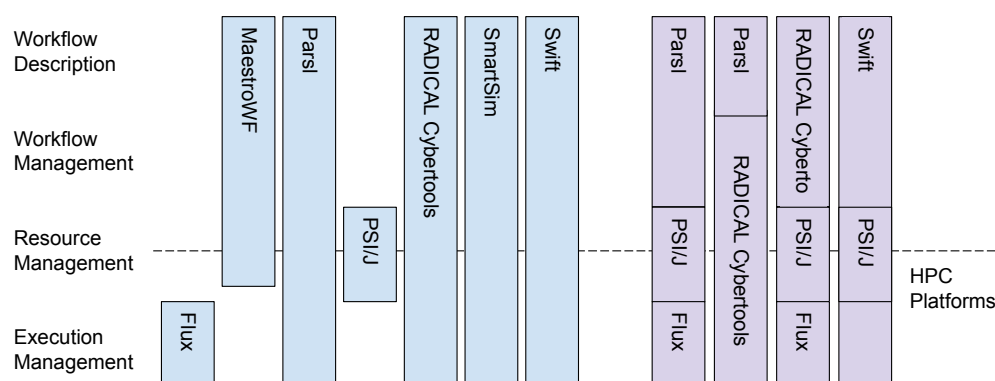
Figure 1. ExaWorks SDK reference stack (right), current components (blue boxes), and examples of integration among components (purple boxes). SDK offers a variety of interfaces, programming models, and runtime capabilities to execute scientific workflows at scale on HPC platforms.

187 can spawn one or more child Flux instances that can manage a subset of the parent's resources while
188 specializing their services, and such a nesting can further recurse.

189   Flux's fully hierarchical, customizable software framework architecture has proven effective on high-end
190 systems, including pre-exascale systems (e.g., LLNL Sierra) as low-level service building blocks for
191 complex workflows (Di Natale et al., 2019). For instance, Flux's flexible design allows users to decide
192 whether or not co-scheduling should be configured and lets users choose their scheduling policies (e.g.,
193 a policy optimized for high job throughput) within the scope of their instance. Further, these workflows
194 can connect to their Flux instance and use Flux's communication primitives such as publish-subscribe,
195 request-reply, and push-pull, as well as asynchronous event handling to facilitate the communication and
196 coordination between co-scheduled jobs via Flux's well-defined, highly portable APIs.

197   **MaestroWF** (Maestro Workflow Conductor) allows users to define multistep workflows and automate
198 the execution of software flows on HPC resources. It uses YAML-based study specifications to describe
199 workflows as directed acyclic graphs that can be parameterized across multiple parameters. Maestro's
200 study specification helps users think about complex workflows in a step-wise, intent-oriented manner that
201 encourages modularity and tool reuse. Maestro runs in user space and does not rely on external services.
202 Maestro is in production use at Lawrence Livermore National Laboratory by a growing user community
203 (several dozen regular users).

204   **SmartSim** is a workflow library that makes it easier to use common Machine Learning (ML) libraries,
205 like PyTorch and TensorFlow, in HPC simulations and applications. SmartSim launches ML infrastructure
206 on HPC systems alongside user workloads and supports most HPC workload managers (e.g., Slurm,
207 PBSPro, LSF). SmartSim also provides a set of client libraries in Python, C++, C, and Fortran. These client
208 libraries allow users to send and receive data between applications and the machine learning infrastructure.
209 Moreover, the client APIs enable the execution of machine learning tasks like inference and online training
210 from within user code. The exchange of data and execution of machine learning tasks is orchestrated by a
211 high-performance in-memory database launched and managed by SmartSim.

212   **Parsl** is a parallel programming library for Python that supports the definition and execution of dataflow
213 workflows. Developers annotate Python programs with decorators, Parsl *apps*, indicating opportunities
214 for asynchronous and concurrent execution. Parsl supports two decorators: PythonApp for the execution
215 of Python functions and BashApp to support the execution of external applications via the command
216 line. Parsl enables workflows to be composed implicitly via data exchange between apps. Parsl supports

217 exchanging Python objects and external files, which can be moved using various data transfer techniques.
218 Parsl programs are portable, enabling them to be moved or scaled between resources, from laptops to
219 clouds and supercomputers. Users specify a Python-based configuration describing how resources are
220 provisioned and used. The Parsl runtime is responsible for processing the workflow graph and submitting
221 tasks for execution on configured resources.

222 Parsl implements a three-layer architecture that makes it amenable to interoperation with other SDK
223 components. Parsl workflows are interpreted and managed by the *DataFlowKernel* (DFK). The DFK holds
224 the dependency graph, determines when dependencies are met, and passes tasks for execution via the
225 Executor interface. The *Executor* interface, implementing Python's `concurrent.Futures` Executor,
226 supports task-based execution, returning a Future instead of results. Parsl has several Executors designed
227 for specific purposes, such as high throughput and extreme scale, several external executors have also been
228 integrated as we describe in §4. Finally, the *Provider* interface is responsible for provisioning resources
229 from different parallel and distributed computing resources. This interface enables integration with PSI/J as
230 a direct replacement for in-built capabilities.

231 **PSI/J** is an API specification for job submission management. One of its primary goals is to provide
232 an API that abstracts access to local resource managers (LRMs), such as Slurm or PBS. PSI/J aims to
233 replace the ad-hoc solutions found in most workflow systems and other software that require portability
234 across multiple HPC clusters. PSI/J comes with a reference Python implementation and an infrastructure
235 for distributed and user-directed testing. That infrastructure enables testing of PSI/J on a wide range of
236 user-accessible platforms, centralizing and making test results publicly accessible. We used a modified
237 version of the PSI/J testing infrastructure for the ExaWorks SDK components.

238 **RADICAL Cybertools (RCT)** are middleware software systems designed to develop efficient and
239 effective tools for scientific computing. Specifically, RCT enables developing applications that can
240 concurrently execute up to 10^5 heterogeneous single/multi-core/GPU/node MPI/OpenMP tasks on more
241 than twenty HPC platforms. Tasks can be implemented as stand-alone executables and/or Python functions.
242 RCT comprises building blocks designed to work as stand-alone systems, integrated among themselves
243 or with third-party systems. RCT enables innovative science in multiple domains, including biophysics,
244 climate science, particle physics, and drug discovery, consuming hundreds of millions of core hours/year.
245 Currently, SDK includes two RCT systems: RADICAL-Pilot (RP) (Merzky et al., 2021) and RADICAL-
246 EnsembleToolkit (EnTK) (Balasubramanian et al., 2016). Both implemented as independent Python
247 modules, RP is a pilot enabled (Turilli et al., 2018; Luckow et al., 2012) runtime system, while EnTK
248 is a workflow engine designed to support the programming and execution workflows with ensembles of
249 heterogeneous tasks.

250 **Swift/T** is a workflow system for single-site workflows. It is based on an automatically parallelizing
251 programming language and an MPI-based runtime. The goal of Swift/T is to efficiently manage workloads
252 consisting of many compute-intensive tasks, such as scientific simulations or machine learning training
253 runs, and distribute them at a fine-grained level across the CPUs or GPUs of the site. The language
254 aspect of Swift/T allows the user to define executions in terms of Python, R, or shell script fragments
255 and then set up a data dependency structure that specifies the order in which data is created. Tasks are
256 executed, possibly including loops, recursive function calls, and other complex patterns. The language
257 has a familiar C or Java-like syntax but automatically provides concurrency within a dataflow control
258 paradigm. Thus, loops and many other syntax features are automatically parallelized. The developer can
259 use the language to launch tasks, manage workflow-level data, and even launch subordinate MPI jobs using

260 multiple mechanisms. Developers can leverage standard mechanisms to use GPUs, node-local storage, and
261 other advanced features.

4 COMPONENT INTEGRATION

262 As seen in §1, the landscape of the software that supports the execution of scientific workflows is fragmented
263 into tools with similar capabilities, often without proper maintenance and support. The SDK contributes
264 to reducing that fragmentation by maintaining a collection of adequately engineered, production-grade
265 workflow tools with a proven track record and promoting integration among those tools. The underlying
266 idea is to avoid reimplementing already available capabilities, instead spending the resources on designing
267 and coding integration layers between the technologies with diverse capabilities. While the idea is simple,
268 its actual implementation is challenging. Software systems designed by independent engineering teams are
269 not thought to be compatible at the abstraction and implementation levels.

270   Our integration experience showed a discrete homogeneity of abstractions among software designed to
271 support scientific workflows. For example, most tools share analogous Task abstractions, assumptions about
272 data dependencies among tasks, and, to some extent, the internal representation of computing resources
273 and how they relate to computing tasks. Most differences were found at the interface and runtime level,
274 where each tool implements distinctive designs and capabilities. SDK tools adopt similar best engineering
275 practices (as most software is designed for production these days), contributing to a certain degree of
276 homogeneity. For example, most SDK tools adopt designs based on well-defined APIs, connectors, and
277 adaptors. Finally, implementation-wise, some tools adopted similar technologies, such as Python and
278 ZeroMQ.

279   On those bases, integration points were clearly to be found at the level of the connectors/adapters and
280 primarily based on translating internal representations of tasks, their data, and their resource requirements.
281 Overall, we could integrate user-facing capabilities with various runtime capabilities, requiring minimal
282 code to be written, mainly to implement translation layers among well-defined interfaces or connectors
283 from existing base classes. Integrating models of resource representations was more challenging, especially
284 concerning resource acquisition capabilities. Sometimes, that required extending existing user-facing
285 interfaces to allow a unified definition of resource requirements, and other times, modifying the data
286 structures and methods used to manage resources at runtime. Even accounting for such code extensions
287 and modifications, integration proved straightforward from both a design and implementation perspective.

288   Overall, the experience gained with the development of SDK shows that to mitigate and reduce
289 fragmentation of the landscape of scientific workflow technologies, tools need to be designed following
290 engineering best practices and established patterns for distributed systems, adopt common abstractions,
291 and utilize common patterns and separation of concerns for the communication and coordination protocols.
292 Perhaps unsurprisingly, by doing all the above, tools are more likely to be able to be integrated with future
293 third-party tools. In the remaining section, we expand this summary by describing the integration between
294 Parsl and RADICAL-Pilot, Parsl and PSI/J, RADICAL-Pilot and Flux, RADICAL-Pilot and PSI/J, and
295 Swift/T and PSI/J. Together, those integrations detail the lesson we learned by delivering SDK.

4.1 RADICAL-Pilot and Parsl Integration

297   Integrating RADICAL-Pilot (RP) and Parsl allows two tools developed by different research groups to
298 work seamlessly together, delivering new capabilities that were not available without their integration. RP
299 and Parsl integration is based on a loosely coupled design, in which RP becomes an executor of Parsl

300 (called RPEX), and both systems send, receive, and share tasks (Alsaadi et al., 2022). RP offers capabilities
301 to concurrently execute heterogeneous (non)MPI tasks across multiple heterogeneous resources, with
302 tasks implemented as an executable or a Python function. Parsl offers workflow management capabilities,
303 allowing the development of data workflows via its API. RP and Parsl capabilities are delivered with no
304 changes to the application code, allowing users to benefit from Parsl's flexible API and RP scale and
305 performance with no code migration or refactoring efforts.
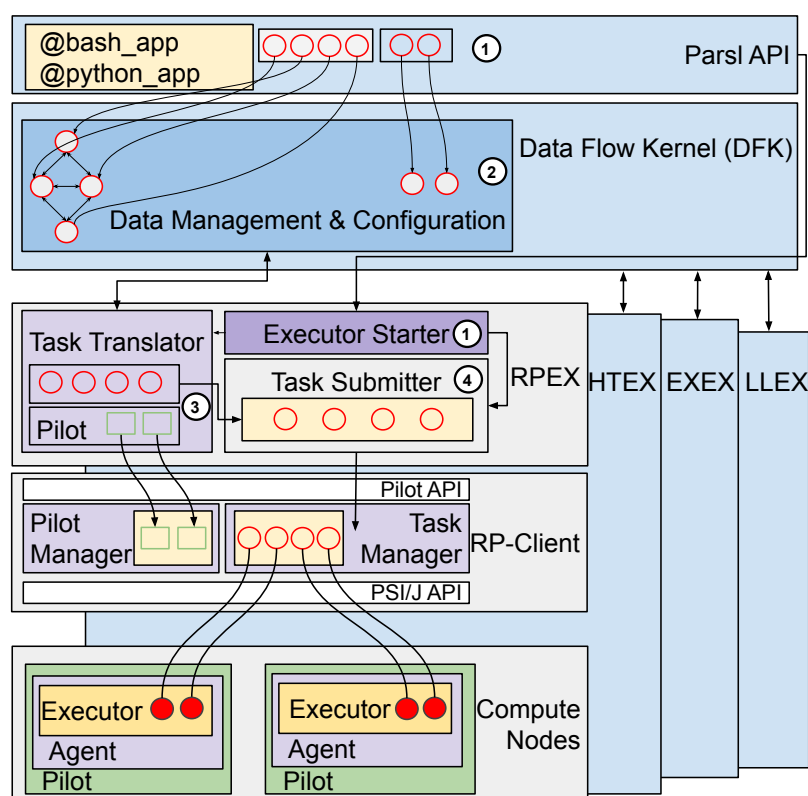
Figure 2. RPEX Architecture. Integration between Parsl (blue boxes) and RADICAL-Pilot (purple and green boxes) via a Task Translator function.

306 RPEX has three main components shown in Fig. 2: executor starter, task translator, and task
307 submitter. Once Parsl starts the executor (Fig. 2 ①), it submits the resolved tasks directly to
308 RP via the Parsl workflow manager as a ready-to-execute task (Fig. 2 ②). Once RP receives
309 the task, it detects the task type, assigns the resource requirements for each task as specified
310 by the user via `parsl_resource_specification` (Fig. 2 ③) and submits the task to RP's
311 `TaskManager` (Fig. 2 ④). Once the tasks are in a state of `DONE, CANCELED` or `FAILED`, RP
312 notifies Parsl about the state of the tasks for further processing or for Parsl to declare the execution as done
313 and to shut down the executor.

314 Integrating RP and Parsl was straightforward except for a difference in resource management. That
315 required aligning RP's task API with Parsl's future tasks by creating a middle point component responsible
316 for translating Parsl's futures into RP's tasks. Most importantly, the task translator's primary duty is to
317 extract the resource requirements from Parsl's tasks and map them to RP's task to enable the use of RP's
318 resource management capabilities in RPEX. Beyond coding RP's interface to Parsl's executor API, we
319 wrote unit tests for RPEX and integrated those tests into Parsl's continuous integration infrastructure.

320 Finally, RPEX required documentation by extending both Parsl and RP documentation and adding specific
321 tutorials.

4.2 Flux Integrations

323 We integrated Parsl and RADICAL-Pilot with Flux to add Flux's scheduling and launching capabilities
324 to both systems' launching methods. As seen in §3, Flux is built to enable effective and efficient execution
325 of large-scale executions of tasks. Both Parsl and RADICAL-Pilot can benefit from those capabilities,
326 especially for homogeneous tasks. Concurrent heterogeneous tasks and high-throughput scheduling require
327 executing multiple Flux instances and partitioning the tasks across those instances.

328 **Parsl**'s FluxExecutor supports Parsl apps that require complex sets of resources (like MPI or other
329 compute-intensive applications) and collections of applications with highly variable resource requirements.
330 Flux's sophisticated and hierarchical scheduling makes these applications logically and efficiently executed
331 across Flux-managed resources. Flux is integrated with Parsl via a Python-based wrapper around the Flux
332 API that implements Parsl's Executor interface. We chose to implement the executor interface rather than
333 Parsl's provider interface, which is used for schedulers, as the executor interface provides more fine-grained
334 control over execution. For example, it allows Flux to manage resources dynamically according to the
335 resource specification provided, rather than the simple batch job provisioning offered by the provider
336 interface. Parsl apps are submitted as Flux Jobs to the underlying Flux scheduler via the Executor API. The
337 Flux executor requires a Flux installation to be available locally and located either in PATH or through an
338 argument passed at creation time.

339 While task execution via the Executor is straightforward, the team had to integrate the Flux resource
340 description model to enable Parsl apps to carry requirements. The integration allows developers to associate
341 a dictionary containing the Flux resource spec with a Parsl app. Supported keys include the number of
342 tasks, cores per task, GPUs per task, and nodes. The Parsl/Flux integration has been used by various users,
343 including for weather modeling workflows at NOAA.

344 The integration process highlighted several challenges, including diverse resource specifications and
345 issues matching environments. As Parsl supports various executors, each with its way of representing
346 resource specifications (e.g., names of attributes, set of configurable attributes provided), we discussed
347 building a common resource specification. We discussed this approach with various Parsl stakeholders and
348 ultimately decided to support the description used by the executor. This simplifies integration significantly
349 but reduces portability between executors as users must convert resource specifications. We based our
350 decision on the fact that, in several cases, there was no one-to-one mapping between attributes, and we were
351 concerned that users familiar with a particular executor would be confused by using different attributes.
352 However, we identify this as an opportunity for future work.

353 After completing the integration, we worked with researchers at NOAA to use it for weather simulations.
354 This work aimed to use Parsl to orchestrate a workflow comprised of diverse task types, from single-core
355 processes to MPI tasks. The most significant obstacle was configuring the environments to work cohesively.
356 Using Spack resulted in issues regarding differing Python dependencies between Parsl and Flux. Ultimately,
357 the team used Spack to deploy Flux, used Pip to install Parsl, and then manually installed two Python
358 libraries to specific versions that worked with both Parsl and Flux. Future work in the SDK will focus on
359 establishing compatible environments and regularly testing the pair-wise installations to identify conflicting
360 versions.

361   **RADICAL-Pilot (RP)** and Flux were integrated using the Flux scheduler and task-launching mechanisms
362   as corresponding components within RP. RP provides component-level API, which allows the creation of
363   different types of RP Scheduler, Executor, and Launching Method. Users can use Flux-based components
364   for a corresponding RP application by specifying it in the target platform configuration. RP starts the Flux
365   instance (i.e., Flux-based scheduler and executor) while bootstrapping its components after having obtained
366   the HPC resources allocated via a pilot job (see Fig. 3). Flux schedules, places, and launches tasks on
367   the compute nodes of that allocation via its daemons. RP tracks task completion and makes available this
368   information across its components. If more tasks are available, RP passes them to Flux to execute on the
369   freed resources. Since RP supports multiple instances of components of the same type, RP can increase
370   the overall task throughput by launching multiple Flux instances within the same job allocation and using
371   them concurrently.
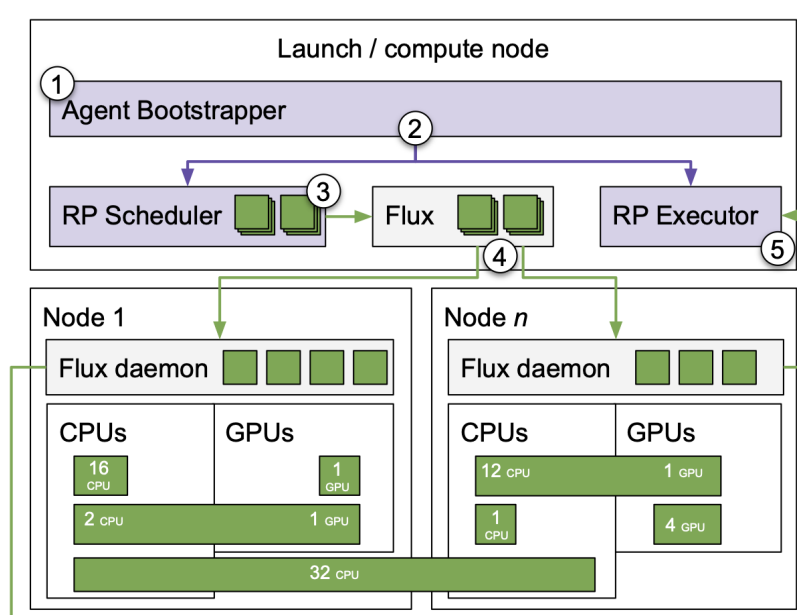
Figure 3. Integration of Flux into RADICAL-Pilot.

372   Internally, RP launches the Flux instances to execute on a specific subset of compute nodes. The Flux
373   instances are configured by adjusting the Slurm environment settings and, specifically, the node list so that
374   each Flux instance 'sees' a different subset of the nodes available in the allocation. Tasks proxied from
375   RP to Flux are then executed on those specific nodes. Task state updates are collected by monitoring the
376   Flux event channel and converting Flux state updates to RP task state updates. Currently, RP implements
377   only basic load balancing between the Flux instances, which can be improved in the future. Flux itself also
378   allows for hierarchical scheduling. In contrast to the approach implemented in RP, Flux hierarchies support
379   only the scheduling of a homogeneous bag of tasks where RP can handle heterogeneous tasks, which is
380   what the workloads that RP typically manages require.

381   The integration of RP and Flux is a significant step towards enabling the execution of large-scale
382   workflows on HPC platforms. The integration is currently being used to support the execution of exascale
383   workflows on the Frontier supercomputer at Oak Ridge National Laboratory.

4.3 PSI/J Integrations

We integrated Parsl, RADICAL-Pilot, and Swift/T with PSI/J, enabling the transparent portability of the applications written utilizing those components across most of the DOE HPC platforms.

Parsl integrates PSI/J by implementing Parsl's extensible *provider* interface in Python (using `psij-python`). The provider interface requires implementing `submit`, `cancel`, and `status` functions, translating to similar calls in PSI/J. These interfaces are very similar as they are both modern and Python-based; thus, integration was relatively straightforward. The main complexity of the integration stems from the need to translate between Parsl job description objects and those used by PSI/J. However, because the data structures are similar, the implementation is minimal, with roughly 150 lines of code, including logging and error handling. However, the benefits of this integration are significant as it enables the Parsl team to leverage the community-maintained interface to the broad ecosystem of schedulers that are used.

RADICAL-Pilot (RP) uses a `PilotLauncher` component to interface with the target resource's batch system, to submit pilot jobs, and to monitor their health. That launcher component traditionally uses RADICAL-SAGA (Merzky et al., 2015) to abstract those interactions. The abstractions provided by PSI/J are conceptually very close to those exposed by SAGA; thus, adding an additional PilotLauncher component based on `psij-python` required only minor changes to RP code. The PSI/J launcher replaced the SAGA launcher for local submissions to the batch system (see "PSI/J API" in Fig. 2). The main semantic difference to the SAGA backend is the support for remote job submission and, thus, for remote pilot placement in RP. However, the PSI/J API itself, and hence the PSI/J-RP integration, is independent of the backend, and the integration code will seamlessly support remote submissions once PSI/J is extended to support those. The SAGA backend will be retired once PSI/J supports remote submissions.

Swift/T has a single-site model, meaning a Swift/T workflow executes inside a single scheduled job. Longer-running campaigns are typically managed through a checkpoint-restart mechanism (see §5 for an example with the CANDLE (Wozniak et al., 2018) use case). Thus, a single scheduler job has to be issued. Constructing this job is a software engineering task that has shared responsibility among three conceptually distinct roles: (1) workflow developer/user, (2) Swift/T maintainer, and (3) HPC site maintainer/administrator. The workflow developer/user simply wants to run scientific workloads using Swift/T and has limited knowledge of Swift/T internals or site-specific technicalities. The Swift/T maintainer understands Swift/T conventions but cannot be responsible for all possible use cases or runtime environments. The HPC site maintainer/administrator installs Swift/T on the site and has in-depth knowledge of the site in question and how to use it. However, it has limited understanding of how Swift/T internals or how its conventions are relevant.

Swift/T was initially packaged with a suite of shell scripts to support running a monolithic Swift/T MPI job using job script templates filtered with settings specified by the user. This model made it easy for advanced users to adjust the scripts as needed quickly. It also posed a code management challenge, as the number of scripts and the variety of computing sites available led to common problems. Swift/T is now bundled with a PSI/J script that fits into this model. Still, all actual interaction with the scheduler is done through PSI/J, pushing the complexity and responsibility of managing scheduler changes and exotic settings into the reusable PSI/J suite. This approach also has the benefit that experienced PSI/J users will likely find Swift/T behavior more understandable.

Swift/T originally used GNU M4 GNU (2024) to filter template scripts into submit scripts for the various HPC workload managers. In this approach, an invocation such as

427 `$ swift-t -m pbs -n 8 workflow.swift`

428 translates `workflow.swift` into internal format `workflow.tic` and issues a PBS job with

429 `$ mpiexec -n 8 workflow.tic`

430 The `-n 8` specifies 8 MPI processes for the job. In practice, there are many other system-related settings
431 that may be provided to `swift-t` via command-line flags and environment variables. Note that the
432 contents of `workflow.swift` do not affect the PBS job; the Swift/T architecture separates the internal
433 logic expressed in the workflow language from the system-level specification of the runtime environment.
434 These settings are then translated by simple M4 patterns into the backend script for PBS specified with
435 `-m pbs`. In the interest of maintainability, we restrict our M4 usage to only `m4_ifelse()` and a custom
436 `getenv()` macro. Much work goes into maintaining these scripts, as the various settings are expressed
437 differently among the workload managers, and site-specific customization is common at supercomputing
438 facilities.

439 In the Swift/T-PSI/J integration effort, we enabled the user to simply specify
440 `$ swift-t -m psij -n 8 workflow.swift`
441 This invokes a wrapper script `turbine2psij.py`, bundled with Swift/T, which is a short Python script
442 that does a `import psij`, then constructs a `psij.JobSpec` with the settings. This is then issued to the
443 workload manager with `psij.JobExecutor.submit()`. Thus, no additional knowledge of workload
444 managers and their specifications is needed on the Swift/T maintainer side. This solves the multi-role
445 software engineering task specified above. The workflow developer/user writes workflows and invokes
446 Swift/T as usual with trivial additional knowledge; any additional knowledge about PSI/J will help with the
447 Swift/T case and other PSI/J uses. The Swift/T maintainer outsources all scripting complexity to the PSI/J
448 maintainers, allowing that role to focus on Swift/T improvements. The HPC site maintainer/administrator,
449 with moderate knowledge of PSI/J and in-depth knowledge of the local site, will be able to easily address
450 any issues with making things work, even without knowledge of Swift/T internals.

5 SUCCESS STORIES

451 We propose three exemplar 'success stories' of using SDK technologies for different use cases in diverse
452 scientific domains. Those show how SDK enables a wide range of workflow applications and resources
453 while offering consistent packaging, testing, and documentation of those technologies. Note that we present
454 use cases that require individual components of SDK but also their integration. That shows how SDK
455 provided an aggregation venue and added value in the form of new capabilities.

456 Note that the choice of a specific workflow technology offered by SDK to satisfy a specific use case is
457 socio-technical. That means that each choice is grounded both on a set of specific capabilities that each
458 tool offers but also on more qualitative factors like ongoing collaborations among research groups and PIs,
459 personal preferences, and present or future funding opportunities. As SDK offers tools with overlapping
460 capabilities, often a use case could be supported by more than a single tool. We see that as a positive factor
461 of the SDK approach, which promotes a software ecosystem with complementary technologies that adhere
462 to high engineering standards, are maintained, well documented, and tested on the target infrastructures.

463 Beyond the three exemplar 'success stories' listed below, SDK components and their integrations are
464 being used to support a variety of ongoing use cases. Those use cases span diverse scientific domains
465 and benefit from the documentation, testing, and capabilities offered by SDK. For example, RADICAL-
466 Cybertools is using the integration with Flux to support the execution of exascale workflows on Frontier.
467 Without Flux, RADICAL-Cybertools would be limited in the number of concurrent tasks that could be

468 executed due to the configuration choices made by the Frontier management team. Further, PSI/J is now
469 the default resource management API for Parsl, RADICAL-Cybertools and Swift and, as such, it is used
470 in every use case supported by those systems. Note that, beyond supporting use cases, Exaworks SDK is
471 now also focusing on expanding the number of workflow technologies supported and establish its testing
472 infrastructure as an official capabilities of several DOE labs.

5.1 Using EnTK for ExaAM workflows
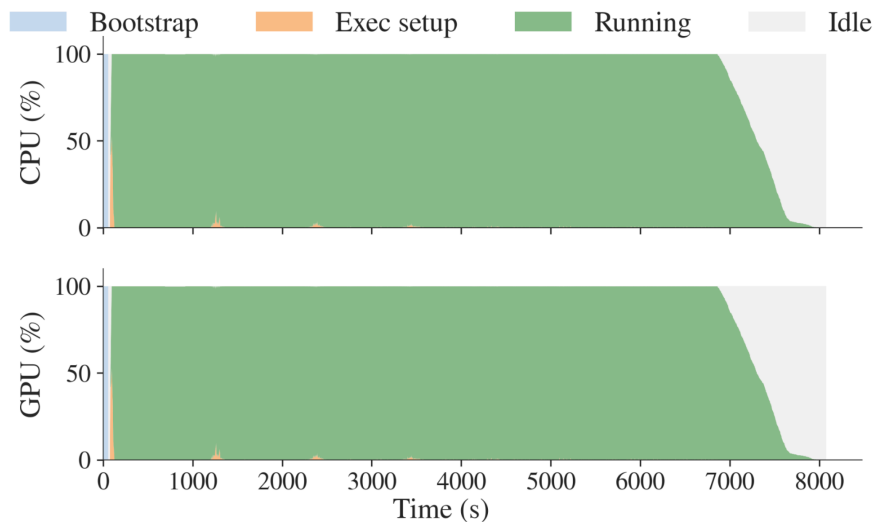
Figure 4. RADICAL Cybertools were used to implement a scalable UQ workflow with the ExaAM team. These plots show overall utilization for the Frontier challenge run was 448,000 CPU cores and 64,000 GPUs, not including 8 CPU cores per node reserved for system processes.

474 The DOE ECP Exascale Additive Manufacturing project (ExaAM) developed a suite of exascale-ready
475 computational tools to model the process-to-structure-to-properties (PSP) relationship for additively
476 manufactured (AM) metal components (Carson et al., 2023). ExaAM built an uncertainty quantification
477 (UQ) pipeline (aka campaign) to quantify uncertainty's effect on local mechanical responses in processing
478 conditions.

479 ExaWorks teamed up with ExaAM to implement a scalable UQ pipeline solution using ExaWorks SDK
480 components. After initial meetings to elicit the ExaAM workflow requirements, the teams selected the
481 SDK and its RADICAL Cybertools component (see §3) to implement that workflow. The details of the
482 ExaAM workflow are presented in (Bader et al., 2023); below, we offer a summary of the implementation
483 and execution of this workflow at scale.

484 ExaWorks collaborated closely with the ExaAM team and user(s) to replicate the existing UQ pipeline
485 and its capabilities using SDK's RADICAL Cybertools and, specifically, its workflow engine called
486 EnTK. EnTK enables expressing workflows as pipelines, each composed of a sequence of stages. Each
487 stage contains a set of tasks, enabling concurrent execution, depending on available resources. EnTK's
488 programming model allowed us to directly implement the UQ pipeline into a set of EnTK pipelines without
489 costly and conceptually difficult translations between different workflow representations.

490 EnTK workflows replaced existing shell scripts that required hands-on management and constant
491 tweaking, made debugging difficult, and consumed resources otherwise available to progress scientific
492 research. SDK and RADICAL Cybertools offered a performant, scalable, automated, fault-tolerant

493 alternative that replicated and enhanced existing capabilities. The developed code is published in the
494 ExaAM project GitHub repository (The ExaAM Project, 2023).

495   We scaled up the EnTK implementation of the ExaAM UQ pipeline on Frontier, utilizing between 40
496 and 8000 compute nodes for between 2 and 4 hours. Fig. 4 shows a utilization plot for the run on Frontier,
497 demonstrating scalability and utilization. Resource utilization reached 90% of the 448,000 CPU cores
498 (Fig. 4(a)) and 64,000 GPUs (Fig. 4(b)) available for a single run, scaling to the whole Frontier.

499 5.2 Enabling CANDLE with Swift/T

500   The Swift/T component of the ExaWorks SDK has been developed throughout the project to make it easier
501 to use Swift/T workflows on exascale computers. As part of the SDK effort, we improved Swift/T packaging
502 by developing integration with Docker containers and more maintainable Spack (Gamblin et al., 2015a)
503 and Anaconda packaging scripts. In this integration, Swift/T was used to run a data analysis workflow
504 called "Challenge Problem: Leave One Out (CPLO)" developed by ECP/AD/CANDLE on Frontier (see
505 Fig. 5). The workflow was previously developed for Summit and ran full scale there, consisting of about
506 5,000 tasks. The workflow task trains the Uno model, a neural network cancer drug response developed
507 at ANL (Wozniak et al., 2020). Each task trains the model on a slightly different subset of the Uno data
508 set; common subsets are trained first, and the model weights are reused in fine-tuning tasks by subsequent
509 tasks, resulting in an expanding tree of training tasks. The goal is to study model performance for each
510 record left out at the leaves of the workflow tree ("Leave One Out"). The workflow was expanded for the
511 transition from Summit to Frontier by breaking up the data set further, resulting in an expansion in the
512 number of child tasks per workflow from N=4 on Summit to N=16 on Frontier, resulting in a new workflow
513 tree of size $18 * 10^6$, a growth factor of 3,600.
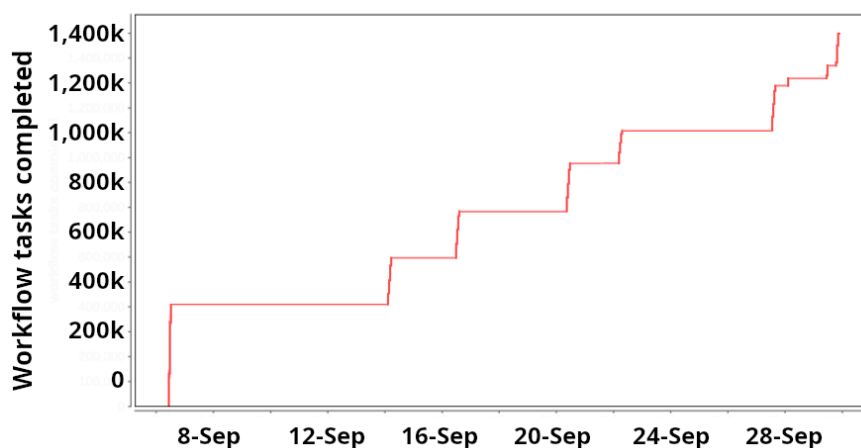
Figure 5. Progress of a typical "Challenge Problem: Leave One Out" campaign implemented for the CANDLE team with the Swift/T ExaWorks SDK component. Several restarts are performed at various scales over a month. During execution, workflow tasks are rapidly completed. This run was used to look for problems in the training data, and only 2 epochs were run per Uno (see the text) task.

514   Multiple changes were needed to run at the expanded scale on Frontier. Most importantly, the training
515 time for the Uno model was significantly shortened on Frontier, necessitating attention to other parts
516 of the workflow, such as its structure plan file and data subset preprocessing. We applied previously
517 developed MPI-IO-based techniques to stage this data to node-local storage before workflow execution.
518 Checkpointing is part of the workflow at the model and workflow task levels; thus, models can be restarted
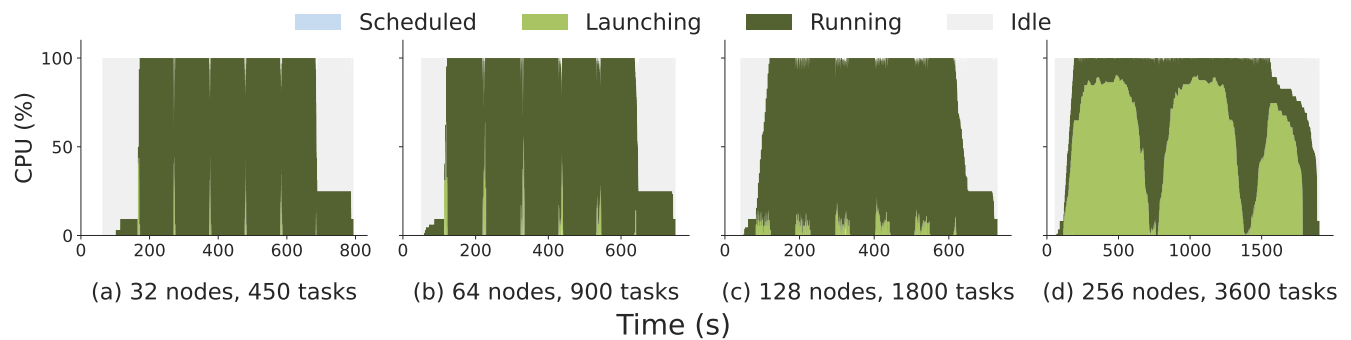
Figure 6. Colmena resource utilization with RPEX (see §4) on 32, 64, 128, and 256 nodes on TACC Frontera with 56 CPU cores per node.

519 from the previously saved epoch (a configurable setting), or, if complete, the whole model is reused, and
520 the workflow task is skipped entirely. We typically run at quarter-system-scale on Frontier, with 1 GPU per
521 Uno model, totaling 8 Uno models per node.

5.3 Colmena with RADICAL-Pilot and Parsl Integration

523 Colmena is a Python package for intelligent steering of ensemble simulations (Ward et al., 2021). Colmena
524 is used to steer large-scale heterogeneous MPI Python functions and executable tasks. Further, Colmena
525 uses Parsl to drive an ensemble of computations on HPC platforms to fit interatomic potentials. Often,
526 Colmena applications require executing MPI computations at a modest scale, which, in turn, requires
527 efficiently running many MPI tasks concurrently. Currently, Parsl executors offer limited MPI capabilities,
528 but SDK provides the integration between Parsl and RADICAL-Pilot, one of the RADICAL Cybertools
529 components (see §4). RPEX, the name given to that integration, satisfies Colmena MPI requirements
530 without requiring changing the existing interface between Parsl and Colmena.

531 RPEX enables Colmena to explore complex multi-physics and multi-scale models by flexibly coupling
532 various types of simulations. This includes seamlessly executing ensembles of tasks with heterogeneous
533 sizes and types, using different computational engines in complex workflows. The provided capabilities
534 are delivered without compromising Colmena's performance and efficiency. We used RPEX in Colmena
535 to execute MPI and Python functions, requiring no changes to the Colmena code base. Fig. 6 shows that
536 RPEX reaches a resource utilization of ∼99% while executing both MPI executables and Python functions
537 on up to 256 compute nodes of TACC Frontera, with 56 cores per node (Alsaadi et al., 2022). While
538 REPEX provides Colmena with new MPI capabilities, RPEX does not introduce additional overheads
539 compared to when Colmena executes only non-MPI functions via Parsl and without RP. In Ref. (Ward
540 et al., 2021), Fig. 3 shows resource utilization comparable to the one achieved with RPEX and showed in
541 Fig. 6.

6 TESTING

542 One of the most challenging problems of testing software on HPC platforms is their heterogeneity: tests
543 on one HPC cluster are not necessarily reproducible on other HPC clusters. This difficulty stems from
544 several reasons, including hardware differences, the use of heavily optimized cluster-specific library
545 implementations, and differences in software and configuration. Specific projects, such as PSI/J (Hategan-
546 Marandiuc et al., 2023), reduce this difficulty by providing a uniform API for accessing parts of the

547 software/library stack. Still, limitations exist to how much one can abstract without compromising
548 performance and required specificity. Thus, a natural solution is to test software on an exhaustive set of
549 HPC platforms. However, such a strategy is generally met with different difficulties since HPC clusters
550 are often found under separate administrative domains with little to no infrastructure that would allow a
551 software package development team to reach a reasonable subset of them. Typically, the result is that the
552 average HPC software package is tested on a small set of platforms that the development team can muster
553 access to, with most testing on other platforms being delegated to users in an ad-hoc fashion. It is important
554 to distinguish here between unit and integration testing. We assume (and encourage, as part of participation
555 in the SDK) that each SDK component contains appropriate unit-level tests.

6.1 Testing Infrastructure Overview

557 The ExaWorks SDK addresses the testing problem by providing an infrastructure that enables testing
558 of the SDK components on DOE platforms and other platforms that users can access. The infrastructure
559 consists of a test runner framework and a dashboard.

560 The test runner framework is a collection of tools and practices that enable the deployment of SDK
561 components and the execution of tests therein. The main drivers for the framework are GitLab Continuous
562 Integration pipelines deployed at various DOE labs. In addition, GitHub actions and direct invocation
563 (which can be driven by the `cron` tool) are also supported and used. Each pipeline consists of a deployment
564 step and a test execution step. Three means of deployment are provided: `pip`, `conda`, and `spack`, but the
565 support varies by the package being tested and the location where the pipeline is run. A location-agnostic
566 version of the SDK is also provided as a `Docker` container. Active pipelines are configured for ALCF/ANL
567 (Polaris), LLNL (Lassen, Quartz, and Ruby), NERSC (Perlmutter), and OLCF/ORNL (Ascent and Summit).
568 The test execution step invokes a set of validation tests for each SDK component and integration tests that
569 validate the correct interfacing of two or more SDK components when appropriate.

570 A testing infrastructure would not be complete without the means to collect and meaningfully present
571 results to developers and (potential) users. The ExaWorks SDK testing dashboard addresses this piece of
572 the testing puzzle: the mechanism to report test results to developers and, in its final production version,
573 also to users. The testing dashboard was initially developed for the PSI/J-Python project, where it enabled
574 the centralization of results of user-maintained test runs. Like PSI/J-Python, the ExaWorks SDK adopts a
575 hybrid approach, with tests on some HPC systems maintained by the ExaWorks team while allowing users
576 to set up test runs on machines under their control.

577 The testing process is as follows. A client-side test runner (typically invoked by a GitLab pipeline)
578 executes desired tests and captures the result and ancillary information, such as output streams or logs.
579 This information is then uploaded to the testing dashboard to present the results to users and developers.
580 By default, results are aggregated by site and date into a calendar view—see Figure 7, which shows a quick
581 overview of the overall pass/fail trends over the last few days. Users then can select sites and navigate to
582 specific test runs, which allows them to examine individual test results and client-provided outputs and
583 logs.

584 The testing dashboard consists of a backend and a frontend. The backend provides authentication,
585 stores test results in a database, and responds to queries for historical test data; the frontend is a web
586 application implemented using the Vue.js (The Vue.js Team, 2024) library and displays the test information
587 to developers and users alike. Like the PSI/J-Python testing dashboard, the SDK dashboard uses a simple
588 authentication mechanism that requires a verified email address. This is done to associate result uploads
589 with an identity that can be used to manage access to the dashboard. The choice to use simple email

Figure 7. The SDK Testing Dashboard Summary Page

590 validation rather than an authentication provider service is motivated by a desire to allow test results from
591 users from any institution and private users.

6.2 Challenges

593 Our approach is not without challenges, some of which are essential to the problem we are attempting to
594 address, while others are consequences of choices we made along the way. One of such difficulties, which
595 results from essential and circumstantial complexities, is that SDK package tests are done with all SDK
596 packages installed. That is because many of the SDK packages are meant to be interoperable, as well as the
597 assumption that in many realistic environments, it is entirely possible or even desirable to have multiple
598 arbitrary SDK packages installed on one system. This can lead to dependency conflicts that individual
599 package maintainers cannot reasonably or effectively address.

600 A related but distinct difficulty is installing diverse packages on heterogeneous computing platforms.
601 In many cases of actual and possible component packages of the SDK, compilation on target compute
602 resources is necessary, with exceptions for Python projects with no native (compiled) parts and other
603 projects written exclusively in interpreted languages. The ExaWorks SDK supports two solutions for
604 compiling and installing packages: Conda (The Conda Team, 2024) and Spack (Gamblin et al., 2015b).
605 Member projects of the SDK must either provide a Conda package specification or be available through
606 Spack (or both). Many projects, however, lack the resources to maintain Conda or Spack packages, and
607 providing such packages as a condition for inclusion in the ExaWorks SDK represents an additional cost.
608 Furthermore, both Conda and Spack are designed to create a sandbox in which specific dependencies
609 are compiled and installed, such as to not conflict with system packages, some of which are optimized
610 specifically for the target system. Although both Conda and Spack allow exceptions to be made and system
611 packages to be used directly, a trade-off is forced: run the risk of incompatibilities by using an optimized
612 dependency provided by the system or sacrifice performance both at run-time as well as during installation
613 by compiling and installing unoptimized dependencies.

614 Finally, testing on multiple HPC platforms managed by diverse organizations poses sociotechnical issues
615 related to supporting those tests on those resources, resource allocation and scheduling, and security
616 policies. Often, in the DOE space, tests need to be audited and approved before being routinely executed on
617 HPC platforms. Further, routinely running tests requires dedicated resources associated with a project and
618 a related allocation. Without specific policies and agreements about resources and allocations dedicated to
619 testing within each institution, ExaWorks runs the SDK tests utilizing the allocation of the project. While

620 that worked for the project's duration, it does not allow the establishment of a long-term testing strategy for
621 SDK within the DOE community.

622 During our work on a testing infrastructure, we encountered several practical obstacles. Our first
623 (chronologically) obstacle was access to HPC systems, especially when a proposed ECP project-wide
624 testbed failed to materialize fully. Our team was left with submitting individual project applications for
625 each HPC center where tests were to be deployed. While we could apply for startup allocations without
626 much effort, none of the centers had provisions for allocations that were meant to address infrastructure
627 testing. Much effort was spent deciphering the required GitLab configurations for each HPC center, with
628 specific and complex configuration files needed for each machine. This could be seen as a success story
629 since individual software package maintainers would have to go through a similar process to run tests on
630 these systems, and the ExaWorks SDK has the potential to tame this difficulty.

7 TUTORIALS AND DOCUMENTATION

631 Alongside testing, documentation is also a fundamental element of the ExaWorks SDK. SDK documentation
632 has to satisfy three main requirements: (1) centralize into a single venue and under a consistent interface all
633 the information specific to SDK; (2) avoid duplication of documentation between SDK and its tools; (3)
634 minimize maintenance overheads; (4) manage a rapid rate of obsolescence in the presence of continuously
635 and independently updated software components; (5) use the same documentation for multiple purposes
636 like training, dissemination, hackathons, and tutorials.

637 Documenting the SDK poses specific challenges compared to documenting a single software system.
638 The SDK is a collection of software components independently developed by unrelated development
639 teams. While each tool is part of SDK, the ExaWorks team does not participate in the development
640 activities supporting each component, decide when and how those components are released, and how each
641 component's documentation is updated or extended. That has the potential to make the SDK documentation
642 obsolete, very resource-intense to maintain, and prone to create duplication detrimental to the end user.

643 We consistently scoped the SDK documentation to offer static and dynamic information. Static
644 information focuses on the SDK itself, offering the needed information about what it is and it is not,
645 the list of current components, the minimum requisites for a component to be part of the SDK, the process
646 to follow to include that component, and details about code of conduct and governance. Due to the
647 challenges listed above, we avoid static information about the SDK components, directly linking specific
648 documentation for each tool. As such, we provide a hub where users can find a variety of pointers to the
649 vast and often dispersed software ecosystem to support the execution of scientific workflows at scale on
650 DOE HPC platforms.

651 Notwithstanding the availability of tool-specific documentation, we had to document the use of those tools
652 on a specific set of DOE platforms and, especially, as a set of integrated systems. Thus, we devised a novel
653 approach to dynamic documentation centered on tutorials designed to be used for outreach, training, and
654 hackathon events. At the core of our approach are Jupyter notebooks containing both documentation and
655 code to deliver: (1) paradigmatic examples of scientific applications developed with the SDK components
656 and executed on DOE platforms; (2) tutorials about capabilities of SDK that serve the specific requirements
657 of the DOE workflow community; (3) detailed examples of resource acquisition, management, tasks
658 definition and scheduling; (4) debugging and tracing workflow executions; and (5) many other common
659 tasks required by executing workflows on HPC platforms.

660   The SDK tutorials avoid overlapping with the tutorials specific to each SDK component, focusing
661   on documenting the use of SDK within the DOE software ecosystem and the integration among SDK
662   components. Such integrations are not specific to any tool, so SDK documentation represents an ideal
663   venue for collecting and organizing that information. Further, SDK tutorials are maintained and distributed
664   via GitHub to make them available to other projects supporting the development of DOE platform workflow
665   applications. GitHub enables contributions from the whole community, creating a single body of encoded
666   information that can be maintained and updated beyond the end of the ExaWorks project.

667   Organizing the dynamic component of SDK documentation on GitHub allowed us to extend its use
668   beyond its traditional boundaries. Utilizing GitHub workflows, we created a continuous integration
669   platform where each Jupyter notebook can be automatically executed every time any SDK component is
670   released. That avoids manually maintaining all the tutorials and makes it immediately apparent when a
671   new component release breaks the tutorial's code. Further, the same tutorials can be seamlessly integrated
672   with Readthedocs (The ExaWorks Project, 2024a), the system we use to compile and distribute the SDK
673   documentation. Executing the tutorials every time the documentation is published guarantees that the
674   tutorials work, greatly improving the quality of the information distributed to the end user. Finally, as part
675   of the GitHub workflows, we package all the tutorials into a Docker container (The ExaWorks Project,
676   2024b) that enables users to execute the tutorials both locally or via Binder (The Binder Project, 2024)
677   with minimal overheads and portability issues.

8 CONCLUSIONS

678   Our experience developing the ExaWorks SDK offers valuable insight. There is a gap in the DOE software
679   ecosystem to support the execution of scientific workflows at scale. On the one hand, many middleware
680   components are required to execute those workflows; on the other hand, DOE maintains diverse HPC
681   platforms with different capabilities, policies, and support levels.

682   Users face the challenge of selecting a set of middleware components to obtain an end-to-end software
683   stack that works on one or more of those resources. That choice is challenging because middleware
684   components have overlapping capabilities, work only on a subset of platforms, may work with some other
685   components, or require the user to lock into a specific software stack. Further, users do not know whether
686   and how well each component is tested, internally (unit tests) and on a specific DOE platform (integration
687   tests). Finally, users must patch together component-specific documentation without tutorials designed
688   to illustrate how to execute workflows at scale on a given DOE platform. Ultimately, users face a steep
689   learning curve and time-consuming testing and debugging of diverse middleware components.

690   As seen in §3, ExaWorks SDK fills that gap by providing a curated set of components that: (1) span the
691   software stack required to execute scientific workflows on DOE HPC platforms; (2) comply with software
692   engineering best practices, including testing coverage, continuous integration, well-defined APIs, and
693   comprehensive documentation; and (3) are maintained, open source and widely adopted. These properties
694   are proven to satisfy the requirements elicited from the DOE workflow community and represent a valuable
695   guideline for future SDKs.

696   Reducing the cost and effort duplication that has produced a slew of comparable software tools with
697   overlapping capabilities requires enabling their integration, independent of the development team that
698   develops and maintains each component. As shown in §4, ExaWorks SDK has proven that such an
699   integrative approach works, requires minimal development effort, and provides added value. §5 detailed

700 how SDK components delivered necessary capabilities to diverse scientific domains, both stand-alone and
701 integrated.

702 Beyond curating and integrating a set of suitable components, the ExaWorks SDK reaffirms the importance
703 of both testing (§6) and documentation (§7). ExaWorks delivered novel approaches and technologies in
704 both domains, showing that continuous integration on the HPC platforms has become necessary. With the
705 growing importance of workflows as a scientific application paradigm, users must be given tangible proof
706 that a middleware stack will not fail them when deployed on a specific HPC platform. Gone are the days
707 when users had the time and resources to work as beta testers or could stand months-long delays before
708 executing a computational campaign. Similarly, documentation is not a useful but an ancillary add-on to
709 the software tools. Building deep stacks of middleware components is becoming increasingly complex,
710 and, again, users do not have the time or resources to 'try it out' and explore the problem space by trial and
711 error.

712 Overall, our experience with SDK indicates that DOE must commit resources to create, maintain,
713 and integrate a curated set of workflow technologies in the long run. The DOE computing facilities
714 must officially support those technologies, and those need to go beyond just the HPC platforms.
715 Continuous integration platforms and git-based repositories integrated with workflow capabilities, alongside
716 virtualization platforms, are becoming necessary components of an HPC platform and require careful design
717 and analysis from a performance point of view, as with any other system of an HPC facility. ExaWorks
718 SDK has paved the way, showing the importance of those capabilities and offering tools, policies, and
719 prototypes that can be readily used and extended. In that direction, follow-up projects like the Partnering
720 for Scientific Software Ecosystem Stewardship Opportunities (PESO) (The PESO Project, 2024) or the
721 Center for Sustaining Workflows and Application Services (SWAS) (The SWAS Project, 2024) are the
722 natural successors of ExaWorks and are already leveraging the work done with the ExaWorks SDK.

ACKNOWLEDGEMENTS

AUTHOR CONTRIBUTIONS

733 This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with
734 the US Department of Energy (DOE). The publisher, by accepting the article for publication, acknowledges
735 that the U.S. Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish
736 or reproduce the published form of the manuscript or allow others to do so for U.S. Government
737 purposes. The DOE will provide public access to these results following the DOE Public Access
738 Plan (http://energy.gov/downloads/doe-public-access-plan). All the authors contributed to discussing

739 and reviewing the structure and content of the paper. Hategan-Marandiuc and Titov wrote §6; Alsaadi,
740 Chard, Merzky, Titov, Turilli, and Wozniak contributed to §3 and §4. Turilli wrote the rest of the paper. SJ
741 edited and also co-organized the paper with Turilli.

REFERENCES

742 Badia Sala RM, Ayguadé Parra E, Labarta Mancho JJ. Workflows for science: A challenge when facing
743   the convergence of HPC and big data. *Supercomputing frontiers and innovations* **4** (2017) 27–47.
744 Ferreira da Silva RF, Casanova H, Chard K, Altintas I, Badia RM, Balis B, et al. A community roadmap
745   for scientific workflows research and development. *2021 IEEE Workshop on Workflows in Support of*
746   *Large-Scale Science (WORKS)* (IEEE) (2021), 81–90.
747 Al-Saadi A, Ahn DH, Babuji Y, Chard K, Corbett J, Hategan M, et al. Exaworks: Workflows for exascale.
748   *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)* (IEEE) (2021), 50–57.
749 [Dataset] WCI T. Workflows community initiative (WCI) (2024).
750 da Silva RF, Badia RM, Bala V, Bard D, Bremer PT, Buckley I, et al. Workflows Community Summit
751   2022: A roadmap revolution. *arXiv preprint arXiv:2304.00019* (2023).
752 [Dataset] Amstutz P, Crusoe MR, Tijanić N, Chapman B, Chilton J, Heuer M, et al. Common Workflow
753   Language, v1. 0 (2016). doi:10.6084/m9.figshare.3115156.v2.
754 Blin MJ, Wainer J, Medeiros CB. A reuse-oriented workflow definition language. *International Journal of*
755   *Cooperative Information Systems* **12** (2003) 1–36.
756 DeLine RA. Glinda: Supporting data science with live programming, guis and a domain-specific language.
757   *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (2021), 1–11.
758 Ristov S, Pedratscher S, Fahringer T. AFCL: An abstract function choreography language for serverless
759   workflow specification. *Future Generation Computer Systems* **114** (2021) 368–382.
760 Korkhov V, Krefting D, Montagnat J, Huu TT, Kukla T, Terstyanszky G, et al. SHIWA workflow
761   interoperability solutions for neuroimaging data analysis. *HealthGrid* (2012), 109–110.
762 Zhang W, Myers A, Gott K, Almgren A, Bell J. AMReX: Block-structured adaptive mesh refinement for
763   multiphysics applications. *The International Journal of High Performance Computing Applications* **35**
764   (2021) 508–526. doi:10.1177/10943420211022811.
765 [Dataset] Kumar P, et al. FerroX massively parallel, 3D phase-field simulation framework (2021).
766 [Dataset] The xSDK Team. xSDK: Extreme-scale scientific software development kit (2017).
767 Heroux MA. Scalable delivery of scalable libraries and tools: How ecp delivered a software ecosystem for
768   exascale and beyond. *arXiv preprint arXiv:2311.06995* (2023).
769 [Dataset] The E4S Project. E4S: A software stack for HPC-AI applications (2024).
770 Rocklin M, et al. Dask: Parallel computation with blocked algorithms and task scheduling. *Proceedings of*
771   *the 14th python in science conference* (SciPy Austin, TX) (2015), vol. 130, 126–132.
772 Ahn DH, Garlick J, Grondona M, Lipari D, Springmeyer B, Schulz M. Flux: A next-generation resource
773   management framework for large hpc centers. *2014 43rd International Conference on Parallel Processing*
774   *Workshops* (2014), 9–17. doi:10.1109/ICPPW.2014.15.
775 Di Natale F, Bhatia H, Carpenter TS, Neale C, Kokkila-Schumacher S, Oppelstrup T, et al. A massively
776   parallel infrastructure for adaptive multiscale simulations: Modeling RAS initiation pathway for cancer.
777   *Proceedings of the International Conference for High Performance Computing, Networking, Storage*
778   *and Analysis* (2019), 1–16. doi:10.1145/3295500.3356197.
779 Babuji Y, Woodard A, Li Z, Katz DS, Clifford B, Kumar R, et al. Parsl: Pervasive parallel programming in
780   Python. *28th ACM International Symposium on High-Performance Parallel and Distributed Computing*
781   *(HPDC)* (2019), 25–36. doi:10.1145/3307681.3325400.

782 Hategan-Marandiuc M, Merzky A, Collier N, Maheshwari K, Ozik J, Turilli M, et al. Psi/j: A portable
783   interface for submitting, monitoring, and managing jobs. *2023 IEEE 19th International Conference on*
784   *e-Science (e-Science)* (2023), 1–10. doi:10.1109/e-Science58273.2023.10254912.

785 Balasubramanian V, Treikalis A, Weidner O, Jha S. Ensemble toolkit: Scalable and flexible execution of
786   ensembles of tasks. *2016 45th International Conference on Parallel Processing (ICPP)* (IEEE) (2016),
787   458–463.

788 Merzky A, Turilli M, Titov M, Al-Saadi A, Jha S. Design and performance characterization of Radical-
789   Pilot on leadership-class platforms. *IEEE Transactions on Parallel and Distributed Systems* **33** (2021)
790   818–829.

791 Partee S, Ellis M, Rigazzi A, Shao AE, Bachman S, Marques G, et al. Using machine learning at
792   scale in numerical simulations with SmartSim: An application to ocean climate modeling. *Journal of*
793   *Computational Science* **62** (2022) 101707. doi:https://doi.org/10.1016/j.jocs.2022.101707.

794 Wozniak JM, Armstrong TG, Wilde M, Katz DS, Lusk E, Foster IT. Swift/T: Large-scale application
795   composition via distributed-memory dataflow processing. *2013 13th IEEE/ACM International*
796   *Symposium on Cluster, Cloud, and Grid Computing* (IEEE) (2013), 95–102.

797 Ahn DH, Bass N, Chu A, Garlick J, Grondona M, Herbein S, et al. Flux: Overcoming scheduling
798   challenges for exascale workflows. *Future Generation Computer Systems* **110** (2020) 202–213. doi:https:
799   //doi.org/10.1016/j.future.2020.04.006.

800 Turilli M, Santcroos M, Jha S. A comprehensive perspective on pilot-job systems. *ACM Computing*
801   *Surveys (CSUR)* **51** (2018) 1–32.

802 Luckow A, Santcroos M, Merzky A, Weidner O, Mantha P, Jha S. P: a model of pilot-abstractions. *2012*
803   *IEEE 8th International Conference on E-Science* (IEEE) (2012), 1–10.

804 Alsaadi A, Ward L, Merzky A, Chard K, Foster I, Jha S, et al. RADICAL-Pilot and Parsl: Executing
805   heterogeneous workflows on HPC platforms. *2022 IEEE/ACM Workshop on Workflows in Support*
806   *of Large-Scale Science (WORKS)* (Los Alamitos, CA, USA: IEEE Computer Society) (2022), 27–34.
807   doi:10.1109/WORKS56498.2022.00009.

808 Merzky A, Weidner O, Jha S. Saga: A standardized access layer to heterogeneous distributed computing
809   infrastructure. *SoftwareX* **1** (2015) 3–8.

810 Wozniak JM, Jain R, Balaprakash P, Ozik J, Collier NT, Bauer J, et al. CANDLE/Supervisor: A workflow
811   framework for machine learning applied to cancer research. *BMC bioinformatics* **19** (2018) 59–69.

812 [Dataset] GNU M4 Web Site (2024).

813 Carson R, Rolchigo M, Coleman J, Titov M, Belak J, Bement M. Uncertainty quantification of metal
814   additive manufacturing processing conditions through the use of exascale computing. *Proceedings of the*
815   *SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage,*
816   *and Analysis* (New York, NY, USA: Association for Computing Machinery) (2023), SC-W '23, 380–383.
817   doi:10.1145/3624062.3624103.

818 Bader J, Belak J, Bement M, Berry M, Carson R, Cassol D, et al. Novel approaches toward scalable
819   composable workflows in hyper-heterogeneous computing environments. *Proceedings of the SC '23*
820   *Workshops of The International Conference on High Performance Computing, Network, Storage, and*
821   *Analysis* (New York, NY, USA: Association for Computing Machinery) (2023), SC-W '23, 2097–2108.
822   doi:10.1145/3624062.3626283.

823 [Dataset] The ExaAM Project. GitHub UQ repository (2023).

824 Gamblin T, LeGendre M, Collette MR, Lee GL, Moody A, De Supinski BR, et al. The Spack package
825   manager: bringing order to hpc software chaos. *Proceedings of the International Conference for High*
826   *Performance Computing, Networking, Storage and Analysis* (2015a), 1–12.

827 Wozniak JM, Yoo H, Mohd-Yusof J, Nicolae B, Collier N, Ozik J, et al. High-bypass learning: Automated
828   detection of tumor cells that significantly impact drug response. *2020 IEEE/ACM Workshop on*
829   *Machine Learning in High Performance Computing Environments (MLHPC) and Workshop on Artificial*
830   *Intelligence and Machine Learning for Scientific Applications (AI4S)* (IEEE) (2020), 1–10.
831 Ward L, Sivaraman G, Pauloski J, Babuji Y, Chard R, Dandu N, et al. Colmena: Scalable machine-learning-
832   based steering of ensemble simulations for high performance computing. *2021 IEEE/ACM Workshop on*
833   *Machine Learning in High Performance Computing Environments (MLHPC)* (Los Alamitos, CA, USA:
834   IEEE Computer Society) (2021), 9–20. doi:10.1109/MLHPC54614.2021.00007.
835 [Dataset] The Vuejs Team. Vue.js (2024).
836 [Dataset] The Conda Team. conda.or (2024).
837 Gamblin T, LeGendre MP, Collette MR, Lee GL, Moody A, de Supinski BR, et al. The spack package
838   manager: Bringing order to HPC software chaos. *Supercomputing 2015 (SC'15)* (Austin, Texas) (2015b),
839   1–12.
840 [Dataset] The ExaWorks Project. ExaWorks: Software Development Kit (2024a).
841 [Dataset] The ExaWorks Project. ExaWorks Software Development Kit Docker Container (2024b).
842 [Dataset] The Binder Project. Reproducible, sharable, open, interactive computing environments (2024).
843 [Dataset] The PESO Project. PESO: Partnering for scientific software ecosystem stewardship opportunities
844   (2024).
845 [Dataset] The SWAS Project. SWAS: Sustaining workflows & application services (2024).