Ambitious Data Science Can Be Painless

Hatef Monajemi[1,2], Riccardo Murri[3], Eric Jonas[4], Percy Liang[5], Victoria Stodden[6], and David Donoho[†,1]

Abstract— **Modern data science research, at the cutting edge, can involve massive computational experimentation; an ambitious PhD in computational fields may conduct experiments consuming several million CPU hours. Traditional computing practices, in which researchers use laptops, PCs, or campus-resident resources with shared policies, are awkward or inadequate for experiments at the massive scale and varied scope that we now see in the most ambitious data science. On the other hand, modern cloud computing promises seemingly unlimited computational resources that can be custom configured, and seems to offer a powerful new venue for ambitious data-driven science. Exploiting the cloud fully, the amount of raw experimental work that could be completed in a fixed amount of calendar time ought to expand by several orders of magnitude. Still, at the moment, starting a massive experiment using cloud resources from scratch is commonly perceived as cumbersome, problematic, and prone to rapid 'researcher burnout.'**

New software stacks are emerging that render massive cloud-based experiments relatively painless, thereby allowing a proliferation of ambitious experiments for scientific discovery. Such stacks simplify experimentation by systematizing experiment definition, automating distribution and management of all tasks, and allowing easy harvesting of results and documentation. In this article, we discuss three such *painless computing stacks*. These include *CodaLab*, from Percy Liang's lab in Stanford Computer Science; *PyWren*, developed by Eric Jonas in the RISELab at UC Berkeley; and the *ElastiCluster-ClusterJob* stack developed at David Donoho's research lab in Stanford Statistics in collaboration with the University of Zurich.

Keywords: Ambitious Data Science, Painless Computing Stacks, Cloud Computing, Experiment Management System, Massive Computational Experiments

I. Introduction

Tremendous increases in the availability of raw computing power in recent years promise a new era in computational science and engineering. Amazon, IBM, Microsoft, and Google now make massive and versatile compute resources available on demand via their *cloud* infrastructure. In principle, it should be possible for an enterprising researcher to individually conduct ambitious computational experiments consuming millions of CPU hours within calendar time scales of days or weeks.

The newly available resources could easily accommodate many thousands of such enterprising research projects. If there were a widespread movement by researchers to pursue this opportunity, it seems we should soon witness the emergence of widespread massive computational experimentation as a fundamental avenue towards scientific progress, complementing traditional avenues of induction (in observational sciences) and deduction (in mathematical sciences) (Hey, Tansley, & Tolle, 2009; Monajemi, Donoho, & Stodden, 2017). Indeed, there have been recent calls and reports by national research and funding agencies such as NSF, NIH, and NRC for the greater adoption of cloud computing and Massive Computational Experiments (MCEs) in scientific research (CSSI, 2019; EACC, 2018; NIH, 2018; NRC, 2013; Rexford, Balazinska, Culler, & Wing, 2018; STRIDES, 2018).

In some fields, this emergence is already quite pronounced. The current remarkable wave of enthusiasm for machine learning (and its deep learning variety) seems, to us, evidence that massive computational experimentation has begun to pay off – big time. Deep neural networks have been around for the better part of 30 years, but only in recent years have they been able to successfully penetrate in certain applications. What changed recently is that researchers at the cutting edge can experiment extensively with tuning such nets and refactoring them. The computational effort to produce annual state of the art AI systems scaled up by more than 300,000x over the last six years (OpenAI, 2018).

With the ability to conduct extensive trial and error searches, dramatic improvements in predictive accuracy over older methods have been found, thereby changing the game. Recent experimental successes of machine learning have disrupted field after field. In machine translation, many major players, including Google and Microsoft, moved away recently from Statistical Machine Translation (SMT) to Neural Machine Translation (NMT) (Lewis-Kraus, 2016; Microsoft Translator, 2016). Similar trends can be found in computer vision (Krizhevsky, Sutskever, & Hinton, 2012; Simonyan & Zisserman, 2014). Tesla Motors predominantly uses deep networks in automated decision-making systems, according to Andrej Karpathy, head

[†] Corresponding Author, donoho@stanford.edu
[1] Dept. of Statistics, Stanford University
[2] Data Science Institute, Stanford University
[3] S3IT, University of Zurich
[4] Dept. of Computer Science, University of California Berkeley
[5] Dept. of Computer Science, Stanford University
[6] School of Information, University of Illinois Urbana-Champaign

TABLE I
FEATURES OF THE PAINLESS COMPUTING STACKS PRESENTED IN THIS PAPER[1]

	ElastiCluster/ClusterJob	CodaLab	PyWren
Scientific Service Layer	Interface	Interface	Framework
Service Type	EMS	EMS	Serverless Execution
Input	Script	Script	Function/Values
Language	Python/R/Matlab	Agnostic	Python
Server Provisioning	Yes	Yes	N/A
Resource Manager	SLURM/SGE	Agnostic	FaaS[3]
Job Submission	Yes	Yes	Yes
Job Management	Yes	Yes	N/A
Auto Script Parallelization[2]	Yes	No	No
Auto Mimic	No	Yes	No
Auto Storage	No	Yes	No
Experiment Documentation	Yes	Yes	No
Reproducible	Yes	Yes	Yes

[1] This table includes available features as of December 2018.
[2] For embarrassingly parallel scripts only. See Section V-C for a definition of an embarrassingly parallel script.
[3] FaaS (Functions-as-a-Service) is a collective name to designate cloud services such as AWS Lambda, Google Cloud Functions, or Azure Functions where server provisioning is hidden from the user and done automatically by the provider.

of AI research[1].

In machine learning problems where computational ambitions have scaled most rapidly, we witness a change not just in the performance of predictive models, but in the scientific process itself. Years ago, the path to progress was typically "use a better mathematical model for the situation." Today's path seems to be: first, to "exploit a bigger database;" and next, to "experiment with different approaches until you find a better way."[2]

Even outside machine learning contexts we can see massive experimentation solving complex problems lying beyond the reach of any theory. Examples include:

- in (Brunton, Proctor, & Kutz, 2016), the authors used data to discover governing equations of various dynamical systems including the strongly nonlinear Lorenz-63 model;
- in (Monajemi, Jafarpour, Gavish, Collaboration, & Donoho, 2013) and (Monajemi & Donoho, 2018), the authors expended several million CPU hours to develop fundamentally more practical sensing methods in the area of Compressed Sensing;
- in (Huang et al., 2015), MCEs solved a 30-year-old puzzle in the design of a particular protein;
- in (Shirangi, 2019) the author conducted 9.5 million reservoir simulations (320,000 CPU-hours) to improve upon state-of-the-art oil field development.

In the emerging paradigm for ambitious data science, researchers may launch and manage historically unprecedented numbers of computational jobs (e.g., possibly even millions of jobs).

Hearing examples of such dramatic (e.g., 10,000x) increases in computational ambition, researchers trained in an older paradigm of interactive 'personal' computing might envision a protracted, painful process involving many moving parts and manual interactions with complex cloud services. Undertaking computations in such fashion at such massive scale would indeed be infeasible.

This paper presents several emerging software stacks that minimize the difficulties of conducting MCEs in the cloud. These stacks offer high-level support for MCEs, masking almost all of the low-level computational and storage details.

II. CLOUD TO THE RESCUE

We have argued that today's most ambitious data scientists now plan projects consuming millions of CPU hours, spread across many thousands of jobs.

Traditional computing approaches, in which researchers use their personal computers or campus-wide *shared* HPC clusters, can be inadequate for such ambitious studies: laptop and desktop computers simply cannot provide the necessary computing power; shared HPC clusters, which are still the dominant paradigm for computational resources in academia today, are becoming more and more limiting because of the mismatch between the variety and volume of computational demands, and the inherent inflexibility of provisioning compute resources governed by capital expenditures. For example, Deep Learning (DL) researchers who depend on fixed departmental or campus-level clusters face a serious obstacle: today's DL demands heavy use of GPU accelerators, but researchers can find themselves waiting for days to get access to such resources, as GPUs are rare commodities on many general-purpose clusters at present.

In addition, shared HPC clusters are subject to *fixed policies* while different projects may have completely different (and even conflicting) requirements. As an example of a policy decision that leads to a unsatisfactory distribution of resources, consider an

[1]Source: a public lecture titled "Software 2.0" by Karpathy, at Stanford's Computer Science department on January 17, 2018.
[2]Even in Academic Psychology! see:Yarkoni and Westfall (2017)

TABLE II
IN-HOUSE HPC CLUSTER VS. CLOUD

Type	Pros	Cons
Cloud	• Immediate on-demand access (no competition with others) • Programmatic access (Infrastructure as Code) • Highly scalable • Highly reliable • Flexible (user chooses cluster configuration) • Users determines policies • No large upfront investment (pay per use) • Complete access as root for software installation • Lower energy consumption per processing hour • No need to buy hardware or space	• Lack of financial incentive in academia (possibly wrongly aligned with budget requirements) • Explicit resource management and vulnerability to overcharge (e.g., shutdown VMs after use) • Data transfer and networking costs (not always fully apparent) • Cloud services APIs may be difficult to use (without EMS)
In-House HPC Cluster	• Usually free or very low cost (often subsidized by universities through overhead tax) • Software stacks are typically managed by dedicated personnel • Data transfer and networking is free • User support included in the service	• Typically not scalable (limited resources) • Long waiting time for resources (competition with others) • Fixed policies as directed by administration • No root access • No financial incentive to provide quality user experience

organization providing clusters with expensive low-latency/high-speed networking. Such an organization would naturally want to maximize the return of its investment; as a result, they may set a fixed policy that incentivizes tightly-coupled parallel jobs to maximize the use of the low-latency network. Consequently, this policy penalizes "embarassingly parallel" workloads and disappoints the users who run such workloads.

On the other hand, the advent of cloud computing offers instant on-demand access to *virtually unlimited computational resources* that can be custom configured to satisfy the needs of individual research projects. Google Cloud Platform (GCP), Amazon Web Services (AWS), Microsoft Azure, and other cloud providers now offer easy access to a large array of virtual machines that cost pennies per CPU hour, making it possible for individual research groups to perform 1 million CPU hours of computation over a few calendar days at a retail cost on the order of 10,000 dollars. The cloud providers also offer access to GPUs in volume for as low as 45 cents/hour/GPU, making them an affordable medium for deep learning research.

The cloud thus offers several advantages over traditional HPC clusters:

- *Scalability and Speed*. With millions of servers spread across the globe, cloud providers today own the biggest computing infrastructures in the world. Therefore, any research group with sufficient research funding can *almost instantly* scale out its computational tasks to thousands of cores without having to wait in a long queue on a shared HPC cluster.
- *Flexibility*. Researchers can adjust the number and configuration of compute nodes depending on their individual computational needs.
- *Reliability*. Public cloud infrastructures were initially built by large IT companies for their own needs and are now relied upon by millions of businesses all over the world for their daily computing needs — they are thus monitored 24/7 and offer excellent uptime and reliability. A good example is Netflix that now operates fully on AWS. In fact, Netflix originally decided to migrate entirely to AWS because the cloud offered a more reliable infrastructure (Izrailevsky, 2016).

Table II provides a more comprehensive comparison of cloud versus in-house HPC clusters.

Despite massive use of the cloud by business, and the cloud's great potential for hosting ambitious computational studies, many academic institutions and research groups have not yet widely adopted the cloud as a computational resource and continue to use personal computers or in-house shared HPC clusters. We believe that much of the in-house computing inertia is due to the perceived complexity of doing massive computational experiments in the cloud. Users schooled in the interactive personal computing model that dominated academic computing in the 1990-2010 period are psychologically prepared to see computing as a very hands-on process. This hands-on viewpoint sees a large computing experiment in terms of the many underlying individual computers, file systems, management layers, files, and processes. Users coming from that background may expect MCEs to require raw unassisted manual interaction with these moving parts, and would probably anticipate that such manual interaction would be very problematic to complete, as there could be many missteps and misconfigurations in carrying out such a complex procedure.

If, truly, the cloud-based experiments involved such manual interaction, the process would at best be exhausting and at worst painful. The many possible problems that could crop up in managing processes manually at the indicated scale would likely be experienced as an overwhelming drag, sapping the experimenter's desire to persevere. Even once the experiment was completed, the burden of having conducted it would likely cast a longer shadow, having drained the ana-

lyst's energy and clarity of purpose, thereby reducing their ability to think clearly about the results.

Summing up, such negative perspectives on cloud-based experiments stem from: the perceived complexity of today's cloud computing interfaces, the perceived difficulty of managing an unprecedentedly large number of computational jobs, and the unmet challenge of ensuring that the results of the experiments can be understood and/or reproduced by other independent scientists.

In addition to obstacles listed above for cloud adoption in academia, there may be other barriers such as apparent lack of financial incentives for cloud use in today's academic funding system, university overhead charges, uncertainty about privacy restrictions, and others, as discussed in detail in Rexford et al. (2018). We hope that such barriers will be removed as more academics adopt the cloud for computational research.

III. Automation in Data Science

Proper automation of computational research activities (Waltz & Buchanan, 2009) offers a compelling way to make massive computational experiments painless and reproducible. In fact the vision dates back more than 50 years.

In particular, in his seminal paper "The Future of Data Analysis," John Tukey (1962) called for the use of automation in data analysis, arguing against the critics of automation (see his Section 17) that:

- Properly automated tools encourage busy data analysts to study the data more
- Automation of known procedures provide data analysts with the time and the stimulation to try out new procedures, and
- It is much easier to intercompare automated procedures.

Tukey could not have foreseen the modern context for such automation, which we now formalize. As we see it, an ambitious data science study involves:

1) *Precise specification* of an experiment, which includes defining performance metrics and a range of systems to be studied.
2) *Distribution, execution, and monitoring* of all the jobs implicitly required in 1).
3) *Harvesting* of all the data produced in 2).
4) *Analysis* of the data collected in 3).
5) *Iterations* of steps (1-4) to run new jobs that may be suggested or required by the results obtained in 4).
6) *Reporting* and dissemination of acquired knowledge.

Additionally, the underlying experiment, to be considered ambitious, may involve either ambitious scale in the data storage, the computations, or both.

As must now be apparent, to operate at an ambitious level, it is crucial to automate all these steps and integrate them seamlessly.

In this article, we describe a few examples of software stacks that facilitate such automation; we call them *Experiment Management Systems* (EMSs). Examples we discuss include CodaLab Worksheets (Liang et al., 2014) and ClusterJob (Monajemi & Donoho, 2015), which are discussed in detail later in the paper. More generally, we use the phrase *Painless Computing Stack* (PCS) to refer to a software stack that abstracts away the difficulties of doing large-scale computation on remote computing infrastructures[3].

As we have already argued in the previous section, unassisted cluster computing (i.e., without EMS assistance) would indeed be painful and draining. Consider a scientist wanting to spread an ambitious workload across multiple shared clusters available via the XSEDE[4] ecosystem. We can envision the scientist using traditional practices quickly becoming frustrated with differences in policies, software environment, choice of scheduler, submission rules, licensing differences (Stodden, 2009), and other requirements for different clusters. Refactoring existing properly working single-processor 'laptop-scale' scripts might also be required, imposing an extra development and source code management burden for the scientist. Finally, merely keeping track of progress on each of several different clusters could be distracting and confusing. Crucially, computational reproducibility is a core requirement of *scientific* data science, because the scientific context requires trust in computational findings and safeguards against possible errors. Ensuring that computations done on a cluster can be reproduced at a later time requires additional important considerations often neglected when manual intervention is involved (Berman et al., 2018; Donoho, Maleki, Rahman, Shahram, & Stodden, 2009; Stodden et al., 2016; Stodden, Seiler, & Ma, 2018). Fortunately, the advent of open-source *container* technologies such as Docker (Merkel, 2014) and Singularity (G. M. Kurtzer, 2017), language-agnostic package managers such as Conda (*Conda*, 2012), and common data platforms such as Google's dataCommons (`https://datacommons.org`) provides a path for facilitating reproducibility in ambitious cloud experiments.

The common vision motivating the development of the several PCS's we describe below (see section V) is that such draining and ultimately confusing demands be abstracted away. This increase in abstraction ought to encourage a proliferation of ever more ambitious experiments, while enabling better clarity of interpretation, better reproducibility, and ultimately better science.

IV. A Taxonomy of Services for Scientific Computing in the Cloud

To better describe how computing stacks presented in the next section interact with cloud infrastructures,

[3] See section IV for a finer-grained classification of these systems.
[4] https://www.xsede.org/ecosystem/resources

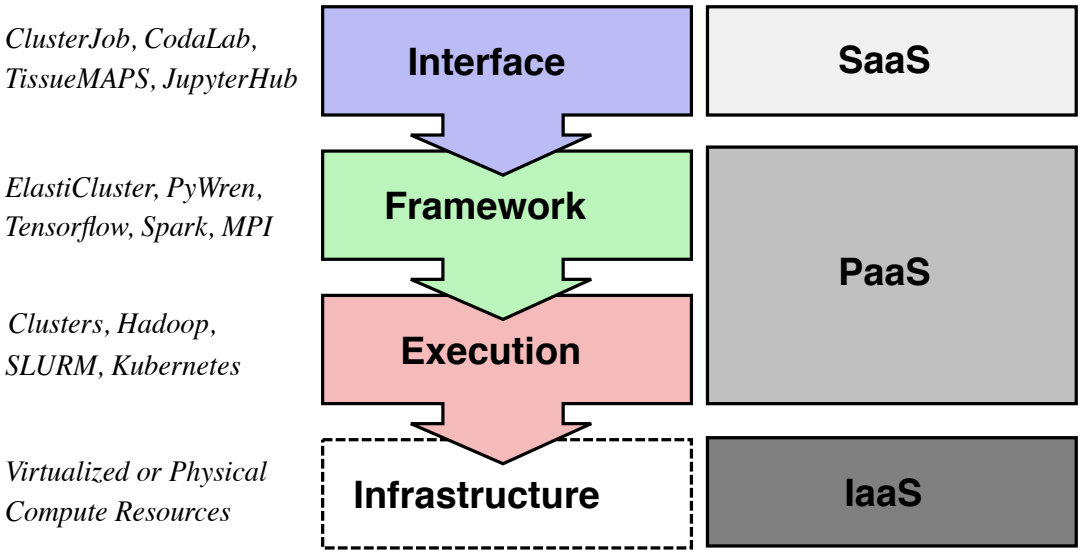

Fig. 1. The layering of services for scientific computing in the cloud (middle) with some examples (left), compared to the NIST classification (Mell & Grance, 2011) of cloud-based IT services (right).

we propose a taxonomy of currently available services for doing scientific computing in the cloud. The reader should keep in mind that the PCS's presented in this article are not necessarily cloud-based, and can well be used in traditional on-premises HPC clusters. We however believe that the coupling of these systems with cloud infrastructures results in greater advantages for scientific research by enabling *very* large computational experiments.

In 2011, NIST introduced (Mell & Grance, 2011) a widely-accepted classification of services offered by cloud providers into three layers:

- *Infrastructure-as-a-Service (IaaS)*: provisioning of compute, storage, networking or other fundamental computing resources. Google Cloud Engine or AWS EC2 are IaaS examples.
- *Platform-as-a-Service (PaaS)*: high-level frameworks and tools to create and run applications on the cloud infrastructure; Google App Engine is a PaaS example that allows developers to easily build and deploy scalable applications without managing cloud infrastructure.
- *Software-as-a-Service (SaaS)*: end-user applications, whose interface (accessed programmatically or through a web client) is tailored to specific tasks. Gmail and Dropbox are familiar SasS examples.

Scientific computing services typically fall under *PaaS* or *SaaS* in the NIST definition; we will introduce a finer-grained classification applicable to scientific computing applications (see Figure 1):

1) *Execution layer:* in our definition, this is the bottom layer and includes services that can take and run a user-provided program, possibly together with some specification of the raw computing resources needed at runtime (e.g., number of CPU cores, amount of RAM). Examples are batch-computing clusters, Hadoop/YARN clusters, container orchestration systems such as Mesos or Kubernetes, and serverless computing services such as AWS Lambda and Azure Functions.
2) *Framework layer:* This layer sits on top of the execution layer and provides users with a way to describe computation in a way that is dictated by an abstract computation model – independent of the raw computing resources actually used. The purpose of the framework layer is to map the abstract computation graph onto a format that can be understood by the execution layer. Examples of software in the framework layer are PyWren [see (Jonas, Pu, Venkataraman, Stoica, & Recht, 2017) and Section V-C later in this paper], Apache Spark (Zaharia, Chowdhury, Franklin, Shenker, & Stoica, 2010), TensorFlow (Abadi et al., 2016), MPI (MPI Forum, 2015; Walker & Dongarra, 1996), and GC3Pie (Maffioletti & Murri, 2012).
3) *Interface layer:* This layer is the topmost layer in our taxonomy and includes services tailored to a specific set of tasks, masking almost all the details of actual computation and storage management. Examples are ClusterJob (Monajemi & Donoho, 2015) (see Section V-A), CodaLab (Liang et al., 2014) (see Section V-B), and TissueMAPS (an integrated platform for large-scale microscope image analysis) (Herrmann, 2017).

V. Painless Computing Stacks

In this section, we present several examples of computing stacks that we consider relatively pain-free for doing large-scale data science studies in the cloud. Table I provides a list of key features for these systems.

In some cases, these systems permit ambitious experiments that otherwise would be inconceivable to

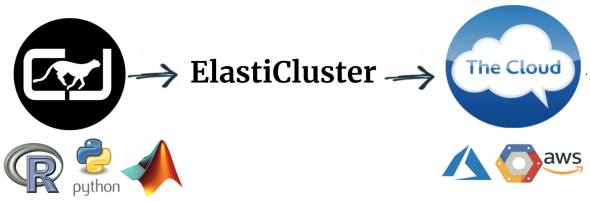

Fig. 2. Elasticluster-ClusterJob stack first provisions a personal cluster in the cloud using ElastiCluster, and then links ClusterJob to this cluster to run ambitious experiments involving many parallel jobs. Experiments are documented reproducibly, and can be retrieved at a later time via ClusterJob interface. This stack is agnostic to the choice of cloud provider and programming language.

conduct. In some other cases, they render experimentation painless, thereby allowing scientists to experiment more. An exact assessment of the extent to which experimentation pain is removed when these stacks are used is beyond the scope of the current article and requires further investigation. Anecdotal evidence from scientists in different disciplines has shown a substantial degree of ease and efficiency in experimental research where these tools are exploited.

A. ElastiCluster-ClusterJob Stack

This stack leverages two different components to conduct massive experiments in the cloud: it first provisions services at the *execution* layer and then exploits services at the *interface* layer to run the actual compute payload. We are focusing in particular on *ElastiCluster* (Murri et al., 2013) to build the virtualized batch-queuing clusters, and *ClusterJob* (Monajemi & Donoho, 2015; Monajemi et al., 2017) to drive the experiments (see Figure 2); we must however emphasize the generality of this model in the sense that the users can use other software systems that offer similar functionalities.

On a more abstract level, this stack includes *building ephemeral clusters* as an additional component of a computational experiment through adopting a Infrastructure as Code (IaC) approach to cloud resource management. Currently, the user makes a call to ElastiCluster to build a cluster, but in the future we expect this step to be handled automatically by ClusterJob.

This stack was first proposed and implemented by Hatef Monajemi and Riccardo Murri during the Stats285 course at Stanford in Fall 2017. Below, we will introduce *ElastiCluster* and *ClusterJob* in more detail.

- **ElastiCluster.** ElastiCluster is an open-source software that provides a command line tool and a Python API to create, set up, and resize compute clusters hosted on IaaS cloud computing platforms. It uses *Ansible* (Red Hat, Inc., 2016) as an IaC tool to get a compute cluster up and running in a push-button way on multiple cloud platforms, such as AWS, Google Cloud, Microsoft Azure, and OpenStack. It offers computational clusters with various base operating systems (e.g., Debian, Ubuntu, CentOS) and job schedulers (e.g., SLURM, SGE, Torque, HTCondor, and Mesos). It also supports Spark/Hadoop clusters and several distributed file systems such as CephFS, GlusterFS, HDFS, and OrangeFS/PVFS. ElastiCluster has been used for large-scale simulation projects such as those occurring in ATLAS experiment at CERN (Haug, Sciacca, & Collaboration, 2017)
- **ClusterJob (CJ).** ClusterJob is an open-source EMS that makes doing massive reproducible computational experiments on remote compute clusters a push-button affair. CJ is written in Perl and currently supports batch submission of Python and Matlab jobs to compute clusters via SLURM and SGE batch-queuing systems. For embarrassingly parallel tasks[5], CJ offers automatic parallelization of scripts that are written serially. In addition, CJ automates reproducibility by generating and saving random seeds for each experiment, list of dependencies, and extra code to ensure the results can be reproduced at a later time. Given a main script and its dependencies, CJ produces a reproducible computational package with distinct *P*ackage *ID*entifier (PID) (a SHA-1 code), automatically sets up the execution environment, and submits the jobs to a remote cluster. Having the PID, one can track the progress of the runs, harvest the data, and get other information about the experiments at any time using various commands provided by CJ's command line interface. At this point, CJ packages are hosted on users' clusters or transferred back to users' local disks. In the next release of ClusterJob, users will have the option of storing their packages on CJHub, a cloud storage that provides automatic archival of the experiments run by CJ.

The ElastiCluster-ClusterJob Stack has been used frequently by Stanford researchers[6]: Romano, Sesia, and Cand'es (2018) conducted eight years of GPU computing to develop and test Deep Knockoffs that improved upon the state-of-the-art variable selection method; Papyan (2018, 2019) reported MCEs conducted using this stack to track the evolution of deep net Hessians; and Mei, Montanari, and Nguyen (2018) used ClusterJob to run thousands of simulations that numerically validated a new mathematical theory for neural networks.

This stack has also been used for teaching data science. Using this stack and cloud computing credits from Google Cloud Education, students of Stanford's Stats285 were able to set up their own personal GPU clusters in the cloud and collectively train nearly 2,000 deep nets with various architectures and datasets in

[5]See Section V-C for a definition of embarrassingly parallel tasks.
[6]In cases where authors do not cite ElastiCluster or ClusterJob, the data is acquired via email correspondence

one calendar day to replicate an important and well-cited article (Zhang, Bengio, Hardt, Recht, & Vinyals, 2016) and discover new phenomena in deep learning [7]. This model has also been used extensively during the 2018 Stats285 Data Science Hackathon[8] to attack challenging problems in political science, medical imaging, and natural language processing. Some of our co-authors regularly use this model of computing. Our experience shows that it takes roughly 15-18 minutes to set up a CPU cluster with less than 10 computational nodes[9] and 20-23 minutes if GPU accelerators are attached to the nodes (an extra five minutes is due to time it takes to install CUDA).[10]

The exact details of this stack are explained thoroughly in the GitHub companion page of this article (`https://tinyurl.com/y2w3yyhp`) (Monajemi et al., 2018). The reader is encouraged to set up a personal cluster following the guide therein. We will briefly explain the general idea here.

An individual can spin up a personal HPC cluster (say `gce`) by providing a simple configuration file to ElastiCluster and typing the following command in a terminal:

```
$ elasticluster start gce
```

Here `elasticluster` is simply a 0-install bash script that is provided to the user. This script uses a dockerized version of ElastiCluster to carry out the user's command; it pulls ElastiCluster's docker image from DockerHub, and then runs elasticluster in a docker container. If Docker is not installed on your machine, the script asks for your permission to automatically install it. ElastiCluster's 0-install script thus brings additional convenience to the user by eliminating the need for the installation of ElastiCluster's API and various dependencies.

Once `gce` cluster is setup, you can run your experiments on it using CJ. All that is needed from your cluster to link it to CJ is the IP address of the frontend (master) node, which can be obtained via the following command:

```
$ elasticluster list-nodes gce
```

To use CJ, one has to install it on a local machine. CJ is written entirely in Perl and features a very straightforward installation guide that is provided in the companion page of this article (Monajemi et al., 2018). Once CJ is available on your machine, you can configure your cluster via either of the following commands:

```
$ cj config gce --update
$ cj config-update gce
```

This command prompts the user to set up the new `gce` cluster by providing the IP address and other optional configuration options such as the desired runtime libraries[11]. The information provided by the user will be saved in CJ's configuration file `~/CJinstall/ssh_config`. For clusters that already exist in this configuration file, a user can update only the corresponding IP address to avoid altering an earlier specification of optional parameters each time a new machine is created.

```
$ cj config-update gce host=35.185.238.124
```

After this step, running MCEs on `gce` is a push-button affair. As a simple example, consider a Deep Learning experiment (written in PyTorch or Tensorflow) that involves training 50 networks for a grid of 10 architectures and 5 datasets. The experimenter first implements a main Python script `DLexperiment.py` that loops over all 50 (*architecture*, *dataset*) combinations and executes a certain task for each. She then includes all additional dependencies including datasets in a directory – say `bin/`.[12]The following CJ command then automatically parallelizes `for` loops inside the main Python script, creates 50 different separate jobs for all the combinations, and reproducibly runs them on `gce` while assigning 1 GPU to each job.

```
$ cj parrun DLexperiment.py gce -dep bin/
 -alloc '--gres:gpu:1' -m 'reminder message'
```

Once the computations associated with certain `<PID>` are finished, the experimenter can harvest the results of all jobs and transfer them to a local machine through various available harvesting commands. As an example, below we reduce all the `results.txt` files of all jobs into one file and transfer the package to the local machine.

```
$ cj reduce results.txt <PID>
$ cj get <PID>
```

The results obtained may then suggest designing and running new experiments, which can be easily handled through CJ. Once all necessary data are collected and the experimenter is satisfied with the current round of experiments, the personal cluster is no longer needed and so it can be destroyed:

[7]The results and discoveries made in Stats285 collaborative study on deep learning are expected to be compiled into a peer-review article.

[8]See course website `http://stats285.github.io`

[9]ElastiCluster currently sets up nodes in batches of 10 at a time. So, for a cluster with N nodes the setup time is roughly $(1 + \lfloor (N-1)/10 \rfloor) \times T$ where T is the setup time for one batch ($T \approx 20$ min).

[10]The time it takes to set up a cluster in the cloud can vary slightly due to various factors such as the proximity and network traffic of the cloud provider's data center, responsiveness of the cloud provider API (e.g., starting a VM on Azure is much more complex than on Google), the boot process of the chosen operating system (e.g., Debian is faster to boot than Ubuntu), the number and speed of CPUs on the local machine, etc.

[11]CJ uses the conda package manager `https://conda.io/docs/` to automatically set up a software environment according to libraries determined in `ssh_config` file.

[12]It is also possible to direct CJ to use datasets already on a cluster, hence not moving data from local machine to remote cluster if data will be used for more than one experiment.

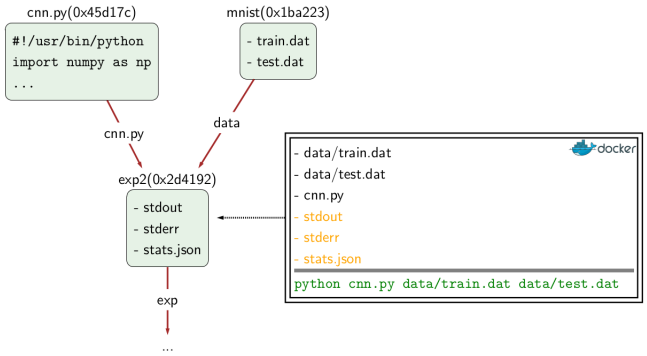

Fig. 3. Execution in CodaLab Worksheets proceeds by taking a set of input bundles (immutable files/directories representing code or data), running arbitrary code in a docker container, and producing an output bundle, which can be used further downstream.

```
$ elasticluster stop gce
```

The information about the computations conducted through CJ is logged and can be retrieved at a later time. CJ provides a very simple command-line interface (CLI) with many features for managing data science experiments. The reader is referred to CJ's documentation available on `www.clusterjob.org` for a comprehensive list of features. It should be emphasized that both ElastiCluster and ClusterJob are open-source software under active development and the reader is encouraged to follow their future enhancements on GitHub.

B. CodaLab Worksheets

CodaLab Worksheets (Liang et al., 2014) offer an EMS developed by a team at Stanford University led by Percy Liang and supported by Microsoft. CodaLab's premise is that in order to accelerate computational research, we need to make it more *reproducible*. Just as version control systems like Git have enabled developers to scale up software engineering, CodaLab hopes to do the same for computational experiments. CodaLab allows users to upload code and data, and run cloud experiments. CodaLab automatically keeps track of the full provenance of computation, so that it is easy to introspect, reproduce, and modify existing experiments.

CodaLab is built around two concepts[13]: *bundles* and *worksheets*. Bundles are immutable files/directories that represent the code, data, and results of an experimental pipeline. There are two ways to create bundles. First, users can *upload bundles*, which are datasets in any format or programs in any programming language. Second, users can create *run bundles* by executing shell commands that depend on the contents of previous bundles. A run bundle is specified by a set of bundle dependencies and an arbitrary shell command. This shell command is executed in a docker container in a directory with the dependencies. The contents of the run bundle are the files/directories which are written to the current directory by the shell command (Figure 3). In the end, the global dependency graph over bundles precisely captures the research process of the entire community in an immutable way.

Worksheets organize and present an experimental pipeline in a comprehensible way, and can be used as a lab notebook, a tutorial, or an executable paper. Worksheets contain references to a subset of bundles and can be thought of as a view on the bundle graph. Worksheets are written in a custom markdown language, and in the spirit of literate programming, allow one to interleave textual descriptions, images, and bundles, which can be rendered as tables with various statistics.

The CodaLab server takes execution requests and assigns jobs to workers. The user can view the results (stdout and any files) in real-time and also communicate with the running process via ports. CodaLab for the foreseeable future will be free to researchers. One unique property about CodaLab is that researchers can pay for their own compute by connecting workers running on Azure/GCP/AWS under their accounts to CodaLab, allowing for more decentralization and larger potential for scaling up organically. They can also install their own CodaLab instance, in which case they supply their own compute and storage.

A CodaLab user can either use the public instance (`worksheets.codalab.org`) or set up a custom instance (e.g., for a research lab). CodaLab can be used either from a web interface or from a command-line interface (CLI), which provides experts with more programmatic control. The CLI is easily installed from PyPI:

```
$ pip install codalab
```

This provides the command `cl`, which is the main entry point to CodaLab functionality.

To upload a bundle (either source code or data):

```
$ cl upload cnn.py
$ cl upload mnist
```

Recall that bundles can either be files or directories. To execute an experiment, one must specify the input bundles (in this case, two of them) and a command to be run, producing a run bundle:

```
$ cl run :cnn.py data:mnist \
  'python cnn.py data/train.dat data/test.dat'
```

For each input bundle, one specifies a name (e.g., `data`). The execution of the command takes place in a Docker image where the input bundles are presented as files/directories with the given names in a temporary directory. The command outputs additional files in the current directory, which are saved as the contents of the run bundle once the bundle finishes executing.

[13]For more information, visit `https://github.com/codalab/codalab-worksheets/wiki`.

The CodaLab execution model is based on dataflow, in which bundles represent information processed in a pipeline. In particular, bundles are immutable, so each command produces a new run bundle rather than modifying an existing bundle. Note that the Docker environment is only used temporarily to run the command; only the outputs of the run are saved. This immutability stands in contrast to other execution models where one might have an entire virtual machine at one's disposal or, in the case of Jupyter notebook, the entire Python kernel. The dataflow model of CodaLab is important for introspection and decentralized collaboration: one can see exactly the chain of commands that were run, and another researcher can build off of an existing result by simply running more commmands on it without a danger of overriding anything. In addition, CodaLab stores all bundles in a bundle store, which is currently based on a file system. Some deployments use local disks and others use a network file system. Access to the bundle store is prohibited, and all access goes through CodaLab APIs, which uses its own group-based permissions model.

One of the most powerful features in CodaLab is called `mimic`, which is enabled by having the dataflow model of computation. In brief, `mimic` allows you to rerun a computation with modifications. The basic usage is as follows:

```
$ cl mimic A B
```

This command examines all the bundles downstream of `A`, and re-executes them all, but now with `B` instead. For example, `A` could be the old dataset and `B` could be the new dataset, or `A` could be the old algorithm and `B` could be the new algorithm. In principle, whenever someone creates a new method, she should be able to painlessly execute it on all existing datasets as long as the new method conforms to a standard interface.

One of the two main uses of CodaLab Worksheets is in running the Stanford Question Answering Dataset (SQuAD) competition (`stanford-qa.com`). In this competition, researchers have to develop a system that can answer factual questions on Wikipedia articles. Over the last two years, over 70 teams have submitted solutions to the highly competitive leaderboard. Each team runs their models on the development set, which is public; this allows teams to independently configure their own environment and manage their own dependencies. Once the team is ready, they submit their system, which is actually the bundle corresponding to the result on the development set. The SQuAD organizers use `cl mimic` to re-run the experiment on the hidden test set instead. Here, we see that CodaLab provides both flexibility and standardization. As a case study, in the past two years, one of the homeworks in the Stanford Natural Language Processing class has been to develop a model for SQuAD. In 2017, 162 teams from the class participated using the public instance of CodaLab, which was able to scale up and handle the load. There are also other leaderboards that are currently supported by CodaLab Worksheets such as QUAC (`quac.ai`), COQA (`stanfordnlp.github.io/coqa`), and MURA (`stanfordmlgroup.github.io/competitions/mura`).

The other main use case of CodaLab is to help people run and manage many experiments at once. This happens at several levels. First, CodaLab is backed by a cluster, so that the user needs to focus only on what experiments to run rather than where to run them (although the user can specify resource requirements). Second, the dataflow model means that for every experiment, which version of the code and data used to produce that experiment is fully documented; as a result, one will never find oneself in a situation with a positive result that is not reproducible anymore. Third, the worksheets in CodaLab offer a flexible way of monitoring and visualizing runs. One can define a table schema, which specifies the custom fields to display (e.g., accuracy metrics, resource utilization, dataset size); a run is a row in this table. This allows one to easily compare the metrics on many variants of the same algorithm, leading to faster prototyping.

In summary, CodaLab provides a collaborative platform that allows researchers to contribute to one global ecosystem by uploading code, data, and other assets, running experiments to generate other assets (bundles), etc. CodaLab keeps the full provenance and provides full transparency (though one can opt to keep some bundles private if necessary). CodaLab starts as a mechanism for enabling researchers to be more efficient at running experiments, and also serves as a publishing platform for published research or competitions. Having a common substrate that supports these use cases opens up the opportunity to bring development and publication closer together.

C. PyWren's serverless execution model

Many scientific computing tasks exhibit a significant degree of innate parallelism, which if properly exploited can dramatically accelerate computational science. These range from classic Monte Carlo methods, to the optimization of hyperparameters, to featurization and preprocessing of large input volumes of data. In many of these cases, large amounts of code are written a priori for one *instance* of such tasks without regard to potential parallel or distributed execution. Indeed, it is only at the end (i.e., the outer per-instance processing loop) that parallelism is even apparent. The code written in this way is called *embarrassingly parallel* because it is trivial to execute in parallel; each operation does not depend on the result of any other. If the computing resources available were truly infinite, the total runtime for these operations would be bounded by the duration of the slowest single scalar piece. Running a single task and running 10,000 tasks would take the same amount of time.

TABLE III
SERVERLESS COMPUTING RESOURCE LIMITS PER INVOCATION

	AWS Lambda[1]	Google Cloud Functions	Azure Functions
Deployment (MB)	50	100	N/A
Memory (GB)	3.0	2.0	1.5
Ephemeral Disk (GB)	0.5	2.0 – *Mem*[2]	5000
Max. run time (sec)	300	540	600

[1] `https://docs.aws.amazon.com/lambda/latest/dg/limits.html`
[2] Disk space consumes from the memory limit.

PyWren (Jonas et al., 2017) is a system developed to enable this kind of massively-parallel, transparent execution. PyWren is built in the Python programming language, and exploits the language's inherent dynamism to transparently identify dependencies and related libraries and marshal them to remote servers for execution. It uses recent *serverless* platforms offered by cloud providers to quickly command controls of tens of thousands of CPU cores, run the resulting parallel task transparently, and then shut down those machines.

Serverless computing [a.k.a. Function as a Service (FaaS)] is a fairly new cloud execution model in which the cloud provider removes much of the complexity of the cloud usage by abstracting away server provisioning (Miller, 2015). In this model, a function and its dependencies are sent to a remote server that is managed by the cloud provider and then executed. AWS Lambda, Google Cloud Functions, and Azure Functions are among popular serverless computing offerings.

Current serverless computing services are suitable for short-lived jobs with small storage and memory requirements because of the limits set by the cloud providers (See Table III). This is because serverless computing was originally designed to execute event-driven, stateless functions (code) in response to triggers such as actions by users, or changes in data or system state. Nevertheless, serverless computing provides an efficient model for applications such as processing and transforming large amount of data, encoding videos, and applications such as simulations and Monte Carlo method with large innate amounts of parallelism (Ishakian, Muthusamy, & Slominski, 2018; Jonas et al., 2017).

PyWren can easily be installed via PyPI and following a number of setup prompts which involve providing credentials for authentication to the underlying cloud computing provider:

```
$ pip install pywren
$ pywren-setup
```

As an example, consider the following `MatVec` function that performs the relatively trivial task of generating a random matrix and vector from a $\mathcal{N}(0, b)$ distribution and computing their matrix-vector product, and returning the result.

```
def MatVec(b):
    x = np.random.normal(0, b, 1024)
    A = np.random.normal(0, b, (1024, 1024))
    return np.dot(A, x)
```

Using PyWren's `map` command, one can painlessly invoke 1,000 distinct instances of this function to be executed transparently in the cloud:

```
pwex = pywren.default_executor()
res  = pwex.map(MatVec,
           np.linspace(0.1, 100, 1000))
```

Behind the scenes, PyWren exploits Python's dynamic nature to inspect all dependencies required by the function, and marshals as many of those as possible over to the remote executor. The resulting function is run on the remote machine, and the return value is serialized and delivered to the client.

The dynamic, language-embedded nature of PyWren makes it ideal for exploratory data analysis from within a Jupyter notebook or similar interactive environment. PyWren is currently limited to exploiting `map`-style parallelism, although active research is underway to broaden the capabilities of the serverless execution model. The function serialization technology is not perfect; currently it struggles with Python modules that have embedded C code, requiring them to be packaged independently as part of a runtime. This too is an active area of research. Finally, the limitations provided by the cloud providers' serverless execution environments (including runtime and memory) constrain the exact functions that can be run, although we anticipate these constraints lessening with time.

D. Third-Party Unified Analytics Interfaces

Several companies provide paid services for painless computing in some third-party cloud; researchers may choose to use their services for conducting their ambitious experiments. A few examples of such companies are Databricks (Databricks, 2013), Domino Data Lab (Domino Data Lab, 2013), FloydHub (FloydHub, 2016) and Civis Analytics (Civis Analytics, 2013). Each of these companies has a slightly different focus (e.g., Databricks focuses on Spark applications whereas FloydHub focuses on Deep Learning) and may use a different computing model for managing computations in the cloud. Nevertheless, all of them build wrappers around cloud services so that individual users can

conduct their MCEs without having to directly interact with the cloud. They provide graphical user interfaces through which users can set up their desired computational environment, upload their data and codes, run their experiments and track their progress. They also offer a community edition of their services that can be used for initial testing before buying their computing and storage services.

VI. Concluding Remarks

We discussed several computing stacks that can dramatically scale up computational experiments, painlessly. Such stacks explicitly or implicitly enable experiment management systems (EMS), a fundamental concept in modern data science research. They offer efficiency and clarity of mind to researchers by organizing the specification and execution of large collections of experiments, and by removing the apparent barriers to using the cloud. In addition, they capture and document the numerous iterative attempts that are undertaken in typical projects.

We look forward to a future where *every* researcher can dream up ambitious computational experiments, open up his/her laptop, and command a computational agent to fire up millions of jobs to study a certain problem of interest. A future where instead of manual human intervention, computational agents seamlessly run jobs in the cloud, manage their progress, harvest the results of the experiments, run specified analyses on those results, and package them in a unified format that is transparent, reproducible, and easily sharable. Such automation of research activities will, we believe, empower data scientists to deliver many more breakthroughs and will accelerate scientific progress.

VII. List of Abbreviations

AWS Amazon Web Services
CJ ClusterJob
EMS Experiment Management System
GCP Google Cloud Platform
HPC High Performance Computing
IaaS Infrastructure as a Service
IaC Infrastructure as Code
MCE Massive Computational Experiment
NMT Neural Machine Translation
PaaS Platform as a Service
PCS Painless Computing Stack
SaaS Software as a Service
SLURM Simple Linux Utility for Resource Management

VIII. Author statement and acknowledgements

This article is based on a series of lectures presented in Fall 2017 at Stanford in Statistics 285. All authors contributed equally.

This work was supported by National Science Foundation grants DMS-0906812 (American Reinvestment and Recovery Act), DMS-1418362 ('Big-Data' Asymptotics: Theory and Large-Scale Experiments) and DMS-1407813 (Estimation and Testing in Low Rank Multivariate Models). We thank Google Cloud Education for providing Stanford's Stats285 class with cloud computing credits. We would like to thank Eun Seo Jo for helpful discussions.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... Isard, M. (2016). Tensorflow: a system for large-scale machine learning. In *Osdi* (Vol. 16, pp. 265–283).

Berman, F., Rutenbar, R., Hailpern, B., Christensen, H., Davidson, S., Estrin, D., ... Szalay, A. S. (2018, March). Realizing the potential of data science. *Commun. ACM*, *61*(4), 67–72. Retrieved from http://doi.acm.org/10.1145/3188721 doi: 10.1145/3188721

Brunton, S. L., Proctor, J. L., & Kutz, J. N. (2016). Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, *113*(15), 3932–3937. Retrieved from http://www.pnas.org/content/113/15/3932 doi: 10.1073/pnas.1517384113

Civis Analytics. (2013). Retrieved from https://new.civisanalytics.com

Conda. (2012). Retrieved from https://conda.io/docs/

CSSI. (2019). *Cyberinfrastructure for sustained scientific innovation (CSSI): Elements and framework implementations.* Retrieved from https://www.nsf.gov/pubs/2019/nsf19548/nsf19548.htm (NSF 19-548)

Databricks. (2013). Retrieved from https://databricks.com

Domino Data Lab. (2013). Retrieved from https://www.dominodatalab.com

Donoho, D. L., Maleki, A., Rahman, I. U., Shahram, M., & Stodden, V. (2009). Reproducible research in computational harmonic analysis. *Computing in Science & Engineering*, *11*(1), 8–18.

EACC. (2018). *Enabling access to Cloud computing resources for CISE research and education (Cloud Access).* Retrieved from https://www.nsf.gov/pubs/2019/nsf19510/nsf19510.htm (NSF 19-510)

FloydHub. (2016). Retrieved from https://www.floydhub.com

G. M. Kurtzer, M. B., V. Sochat. (2017). Singularity: Scientific containers for mobility of compute. *PLoS ONE*, *12*(5).

Haug, S., Sciacca, F. G., & Collaboration, A. (2017, oct). ATLAS computing on Swiss

Cloud SWITCHengines. *Journal of Physics: Conference Series*, *898*, 052017. Retrieved from https://doi.org/10.1088%2F1742-6596%2F898%2F5%2F052017 doi: 10.1088/1742-6596/898/5/052017

Herrmann, M. D. (2017). *Computational methods and tools for reproducible and scalable bioimage analysis* (Unpublished doctoral dissertation). Universität Zürich.

Hey, T., Tansley, S., & Tolle, K. (2009). *The fourth paradigm: Data-intensive scientific discovery*. Microsoft Research. Retrieved from https://www.microsoft.com/en-us/research/publication/fourth-paradigm-data-intensive-scientific-discovery/

Huang, P., Feldmeier, K., Parmeggiani, F., Fernandez-Velasco, D. A., Höcker, B., & Baker, D. (2015). De novo design of a four-fold symmetric TIM-barrel protein with atomic-level accuracy. , *12*, 29–34.

Ishakian, V., Muthusamy, V., & Slominski, A. (2018). Serving deep learning models in a serverless platform. *arXiv:1710.08460*.

Izrailevsky, Y. (2016). *Completing the Netflix Cloud migration.* Retrieved from https://techcrunch.com/2015/11/24/aws-lamda-makes-serverless-applications-a-reality/

Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., & Recht, B. (2017). Occupy the Cloud: Distributed computing for the 99%. *arXiv:1702.04024*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*.

Lewis-Kraus, G. (2016). *The great A.I. awakening.* Retrieved from https://www.nytimes.com/2016/12/14/magazine/the-great-ai-awakening.html

Liang, P., et al. (2014). *Codalab worksheets: Accelerating reproducible computational research.* Retrieved from http://worksheets.codalab.org

Maffioletti, S., & Murri, R. (2012). GC3PIE: a python framework for high-throughput computing. In *Egi community forum 2012/emi second technical conference* (Vol. 162, p. 143).

Mei, S., Montanari, A., & Nguyen, P.-M. (2018). A mean field view of the landscape of two-layer neural networks. *Proceedings of the National Academy of Sciences*, *115*(33), E7665–E7671. Retrieved from https://www.pnas.org/content/115/33/E7665 doi: 10.1073/pnas.1806579115

Mell, P., & Grance, T. (2011). *The NIST Definition of Cloud Computing.* Retrieved from https://csrc.nist.gov/publications/detail/sp/800-145/final doi: 10.6028/NIST.SP.800-145

Merkel, D. (2014, March). Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, *2014*(239). Retrieved from http://dl.acm.org/citation.cfm?id=2600239.2600241

Microsoft Translator. (2016). *Microsoft translator launching neural network based translations for all its speech languages.* Retrieved from https://blogs.msdn.microsoft.com/translation/2016/11/15/microsoft-translator-launching-neural-network-based-translations-for-all-its-speech-languages/

Miller, R. (2015). *AWS Lambda makes serverless applications a reality.* Retrieved from https://techcrunch.com/2015/11/24/aws-lamda-makes-serverless-applications-a-reality/

Monajemi, H., & Donoho, D. L. (2015). Clusterjob: An automated system for painless and reproducible massive computational experiments. Retrieved from https://github.com/monajemi/clusterjob

Monajemi, H., & Donoho, D. L. (2018). Sparsity/undersampling tradeoffs in anisotropic undersampling, with applications in mr imaging/spectroscopy. *Information and Inference: A Journal of the IMA*, iay013. Retrieved from http://dx.doi.org/10.1093/imaiai/iay013 doi: 10.1093/imaiai/iay013

Monajemi, H., Donoho, D. L., & Stodden, V. (2017, February). Making massive computational experiments painless. *Big Data (Big Data), 2016 IEEE International Conference on*.

Monajemi, H., Jafarpour, S., Gavish, M., Collaboration, S. C. ., & Donoho, D. L. (2013). Deterministic matrices matching the compressed sensing phase transitions of Gaussian random matrices. *PNAS*, *110*(4), 1181–1186. Retrieved from http://www.pnas.org/content/110/4/1181.abstract doi: 10.1073/pnas.1219540110

Monajemi, H., et al. (2018). *Companion page for the paper* Ambitious Data Science Can be Painless. Retrieved from https://monajemi.github.io/datascience/pages/painless-computing-models

MPI Forum. (2015). *MPI: A Message-Passing Interface Standard, Version 3.1.* Retrieved from https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

Murri, R., et al. (2013). ElastiCluster. Retrieved from https://github.com/gc3-uzh-ch/elasticluster

NIH. (2018, Sept). NIH strategic plan for data science. *NIH*, 1–31. Retrieved from https://datascience.nih.gov/sites/default/files/NIH_Strategic_Plan_for_Data_Science_Final_508.pdf

NRC. (2013). *Frontiers in massive data analysis*. The

National Academies Press. Retrieved from https://www.nap.edu/catalog/18374/frontiers-in-massive-data-analysis doi: 10.17226/18374

OpenAI. (2018). *AI and Compute.* Retrieved from https://blog.openai.com/ai-and-compute/

Papyan, V. (2018). The full spectrum of deep net Hessians at scale: Dynamics with sample size. *arXiv:1811.07062*.

Papyan, V. (2019). Measurements of three-level hierarchical structure in the outliers in the spectrum of deepnet Hessians. *arXiv:1901.08244*.

Red Hat, Inc. (2016). *Ansible is simple IT automation.* Retrieved from https://www.ansible.com/

Rexford, J., Balazinska, M., Culler, D., & Wing, J. (2018). *Enabling computer and information science and engineering research and education in the Cloud* (Tech. Rep.). USA.

Romano, Y., Sesia, M., & Cand'es, E. J. (2018). Deep knockoffs. *arXiv:1811.06687*.

Shirangi, M. G. (2019). Closed-loop field development with multipoint geostatistics and statistical performance assessment. *Journal of Computational Physics*, *30*, 249–264.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*.

Stodden, V. (2009). The legal framework for reproducible scientific research: Licensing and copyright. *Computing in Science Engineering*, *11*(1), 35–40. doi: 10.1109/MCSE.2009.19

Stodden, V., McNutt, M., Bailey, D. H., Deelman, E., Gil, Y., Hanson, B., ... Taufer, M. (2016). Enhancing reproducibility for computational methods. *Science*, *354*(6317), 1240–1241. Retrieved from http://science.sciencemag.org/content/354/6317/1240 doi: 10.1126/science.aah6168

Stodden, V., Seiler, J., & Ma, Z. (2018). An empirical analysis of journal policy effectiveness for computational reproducibility. *Proceedings of the National Academy of Sciences*, *115*(11), 2584–2589. Retrieved from http://www.pnas.org/content/115/11/2584 doi: 10.1073/pnas.1708290115

STRIDES. (2018). *The science and technology research infrastructure for discovery, experimentation, and sustainability (STRIDES) initiative.* Retrieved from https://commonfund.nih.gov/strides (NIH)

Tukey, J. W. (1962, 03). The future of data analysis. *Ann. Math. Statist.*, *33*(1), 1–67. Retrieved from https://doi.org/10.1214/aoms/1177704711 doi: 10.1214/aoms/1177704711

Walker, D. W., & Dongarra, J. J. (1996). MPI: A standard message passing interface. *Supercomputer*, *12*, 56–68.

Waltz, D., & Buchanan, B. G. (2009). Automating science. *Science*, *324*(5923), 43–44. Retrieved from http://science.sciencemag.org/content/324/5923/43 doi: 10.1126/science.1172781

Yarkoni, T., & Westfall, J. (2017, Nov). Choosing prediction over explanation in psychology: Lessons from machine learning. *Perspect Psychol Sci*, *12*, 1100–1122.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. *HotCloud'10 Proc. 2nd USENIX Conf. on Hot Topics in Cloud Computing*.

Zhang, C., Bengio, S., Hardt, M., Recht, B., & Vinyals, O. (2016). Understanding deep learning requires rethinking generalization. *arXiv:1611.03530*.