The Case for Experiment-Oriented Computing

Paulo Salem
Dell EMC
São Paulo, Brazil
paulosalem@paulosalem.com

ABSTRACT

Experimentation aspects (e.g., systematic observation, exploration of alternatives, formulation of hypotheses and empirical testing) can be found dispersed and intertwined in many software systems. Numerous examples are provided in this article. This suggests that experimental activity is a class of computation in its own right. By abstracting the relevant experimentation features, a general *Experiment-Oriented Computing* (EOC) approach, orthogonal to other Software Engineering issues, is formulated in this article. Through this separation of concerns, it is possible to clearly pursue both theoretical and applied research with respect to experimental aspects. Concrete directions for such research and development are also given to illustrate the value of the approach.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; **Experimentation**; • **Software and its engineering** → *Software architectures*; Software verification and validation; • **Computing methodologies** → Machine learning; Modeling and simulation;

KEYWORDS

experimentation, empiricism, data science, scientific discovery, autonomous agents

ACM Reference Format:
Paulo Salem. 2018. The Case for Experiment-Oriented Computing. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3236024.3264831

1 INTRODUCTION

Systematic observation, formulation of hypotheses and empirical testing, among other methodological processes, are at the heart of modern science. Many software (and hardware) systems have similar concerns, which suggests that scientific experimentation can be seen as a form of computation. Having this in mind, an *experiment* can be broadly defined as a computation in which an *experimental setup*, under the control of an *experimenter*, interacts with autonomous *subjects* in order to extract information for a particular *purpose*. Figure 1 illustrates the software architecture and workflow thus induced. The present article develops this way of understanding computation and its practical realization.

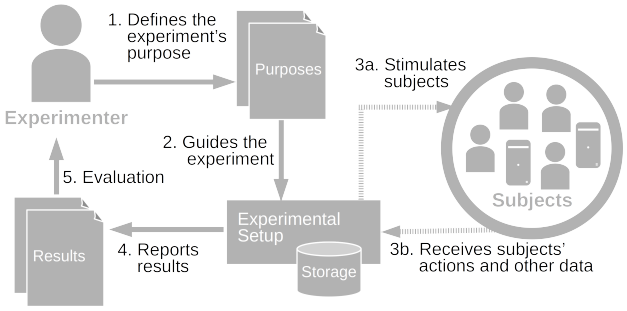

Figure 1: The EOC architecture, including various separation of concerns and the resulting workflow (steps 1 to 5). Subjects can be anything with which interaction is possible (e.g., people, other computing systems). Results may include verdicts, statistics or other structures, both for the appreciation of the experimenter and for reuse in later experiments.

In Software Engineering, perhaps the most traditional area in which experimentation aspects are present is Software Testing [1]. This becomes clear by carefully stating the problem of interest: a certain subject (system under test) undergoes a number experiments (test executions) in order to check, empirically, whether certain properties (test cases) hold or not. Testing engines, therefore, are rather general experimentation technologies, although they are usually employed only for software verification.

Such experimentation concerns are widespread in Computer Science, although often intertwined with other matters and thus not fully appreciated in their own right. This text provides ample evidence of this (Section 2) and proposes supporting experimentation principles that can be abstracted (Section 3). In this manner, *Experiment-Oriented Computing* (EOC) is established as a distinct way of designing computing systems.

The importance of highlighting such common aspects is not only theoretical, it has historical precedents. For instance, concurrent programming could only be properly *formulated* and *addressed* by recognizing concurrency as a separate aspect of systems [17]. The case for experimental features of systems is similar. Hence, technological development and research directions, as well as some examples of practical software that could arise from this understanding, concludes the exposition (Section 4). The article aims both at emphasizing the increasing importance of experimentation engineering and at driving its progress through a unified conceptual foundation.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264831

2 EXPERIMENTATION IN PRACTICE

The following examples have been selected to provide a somewhat wide perspective on the matter, although they do reflect the author's biases and interests. Nevertheless, these should be sufficient to convince the reader of the pervasiveness of experimental aspects and motivate his or her own observations.

Software Testing. As pointed out in Section 1, Software Testing [1] is an exemplary case of computational experimentation, which can be readily seen under the architecture of Figure 1.

Formal Verification. Formal Verification consists in determining whether a system (like Software Testing) or model (unlike Software Testing) satisfy a set of properties. Such properties are defined in special mathematical notation (e.g., temporal logics, automata, expressions in process algebras), which varies according to the verification technique employed (e.g., Model Checking [2], bisimulation verification). Like in traditional scientific methodology, hypotheses are formulated and evaluated, although in these different notations. However, verification is purely mathematical, not empirical.

Model-Based Testing (MBT), Runtime Verification (RV). Common ground exists between these two broad verification areas. Model-Based Testing [20, 44] consists in using abstract models of systems in order to generate test cases such that the system can be tested with respect to certain criteria. In Runtime Verification [7], on the other hand, no test cases are necessary, one monitors the actual executions of systems in order to continually check formally specified properties.

Discrete-Event Simulation (DES). Simulations can be seen as the manipulation of autonomous experimental subjects. For example, the execution and analysis of simulations can be directly expressed in experimentation terms: the properties to be satisfied are the (formally specified) hypotheses of an experimenter, the simulated system (defined through models, formulas or programs) is the subject under experimental investigation [6, 37]. Optimization of simulated systems is yet another application [15].

Search-Based Software Engineering (SBSE). Many Software Engineering problems can be cast in terms of optimization procedures, an observation that created the field of Search-Based Software Engineering [18]. Because this usually requires heuristic searches, the approach can be understood as trial and error experimentation. Furthermore, some SBSE techniques do require interaction with truly autonomous subjects, as the next example shows.

Automated User Interface Design. User interfaces (UIs) can be automatically generated, evaluated and optimized. Approaches differ with respect to which and how these activities are automated. Of critical importance here is that some do not require actual user interaction for generation [12], while others do [24, 36]. In all of them, however, human ingenuity (in crafting the building blocks of UIs or the user experiments) is combined with algorithmic techniques (during execution).

Distributed Systems. Experimental profiling of distributed systems has been proposed as a way to evaluate infrastructure providers [39] and actively disrupting production clusters is used in practice to ensure their fault-tolerance [3].

Machine Learning (ML), Data Mining (DM), Data Science (DS). The whole field of Machine Learning can be understood from an experimentation perspective, since it concerns the formulation of hypotheses that best explain experiences [30]. In industry, the derived fields of Data Mining [45] and, more recently, of Data Science [9] are also closely related.

Scientific Discovery. Artificial Intelligence (AI) often depends on empirical methods. The work of of Langley *et al.* [25], in particular, establishes a direct relationship between computing and traditional scientific practice to investigate the computational aspects of *scientific discovery*. Džeroski *et al.* [11] provide more recent results on the topic. The "robot scientists" of Sparkes *et al.* [41] demonstrate how such discovery systems can be augmented with physical interaction capabilities, hence allowing fully autonomous biomedical experimentation.

Reinforcement Learning. Both supervised and non-supervised Machine Learning methods typically assume data as given. From the point of view of *engineering* experiments, a more interesting approach is that of Reinforcement Learning (RL) [42], in which agents are supposed to interact with an environment in order to generate the necessary data. This emphasizes the fact that acquiring information is a challenge in its own right.

Causality. One critical objective of many traditional scientific experiments is to determine causal relations. In Artificial Intelligence (AI), this matter has been extensively studied, notably through Bayesian networks. General computational understanding of causality is now available [32].

Black-Box Optimization. In optimization problems, black-box approaches such as Auto-WEKA [43] and Google Vizier [14] can be understood as experiments as well. The latter is a particularly interesting example, because it provides not only the necessary algorithms, but also a sophisticated software infrastructure to support the entire workflow.

Empirical Software Engineering. The field [40] consists in using experimental techniques to *assess* the characteristics of software artefacts and engineering practices. By contrast, in EOC one is interested in *building* systems that perform experiments. Naturally, one can imagine an EOC system that supports Empirical Software Engineering.

3 SUPPORTING PRINCIPLES

The main elements of an EOC architecture, presented in Figure 1, are rather straightforward and intuitive. The discussion above, moreover, shows many ways in which experimentation capabilities are found dispersed in current computing technology. In the present section, supporting principles are abstracted from these existing approaches, in order to provide a unified foundation for such capabilities. These principles are not claimed to be exhaustive, but simply important and of sufficient generality. Table 1 summarizes these relationships. It is also indicated to which elements of the architecture given in Figure 1 each principle mainly applies.

Table 1: A name or reference in a cell exemplifies how the principle (column) can be realized in a field (row). If the principle is present but no particular name or reference is appropriate (e.g., because it is too fundamental), a ✓is used instead. The table is not exhaustive and is aimed merely at illustrating the cross-sectional character of the principles, as well as to suggest potential complementarities among different fields. The latter is best understood by imagining how to fill in the blanks.

Field	Hyp. Form.	Optm.	Expl.	Coord.	Nondet.	Reprod.	Traceab.	Durab.	Fault-Tol.	Lab.
Soft. Test.	test cases					✓	test case granularity			IDE support
Formal Verif.	formal spec.			[19, 29]	[19, 29]	✓	spec. granularity			[4]
MBT, RV	formal spec.	coverage criteria	[10]		[20]	✓	[26]			[5]
SBSE		✓	✓					software		
DES	[6, 37]	[15]	✓	[37]	[37]	✓				[6]
Aut. UI Design	[31]	[12, 24, 31, 34, 36]	[34]	user interac.	[36]			generated UI		
Dist. Sys.			[39]	[13]				data replication	[3, 39]	
ML, DM, DS	model validation	model fitting	exploratory data analysis			notebooks [23]		model persistence		notebooks [23]
Scient. Disc.	✓		✓	[41]		✓				
Reinf. Learn.		rewards	action selection					learning	✓	
Causality	✓						causal graphs			
Black-Box Optim.		cost function, auto. model selection [43]						[14]	[14]	[14]

Hypothesis formulation (1 in Figure 1) and evaluation (2, 3 in Figure 1). Experimentation often concern the formulation of questions to be answered. Computationally, these can take the form of logical formulas, automata, process algebra expressions, test cases or Machine Learning models. Such hypotheses must then be evaluated, which may be accomplished by formal verification, simulation, model validation or mere program execution. The division of subjects in treatment and control groups is relevant when causal relations are sought [32].

Optimization (1, 2, 3 in Figure 1). Although traditional scientific experimentation is typically given in terms of hypothesis formulation and testing, this is by no means the only way to think about experiments. For instance, the purpose of an experiment might be the minimization of the error of some target function, instead of assessing the truth value of a hypothesis.[1]

Exploration (1, 2, 3 in Figure 1). Accumulation of general knowledge about subjects can be a central purpose. Application profiling is one such case. Random sampling of execution paths in a large state-space is one example of exploration technique that can be used in multiple applications [10].

[1]Nevertheless, some optimizations may be understood also as sequences of increasingly better hypotheses.

Coordination (Figure 1). The overall experiment can be seen as the coordination among an experimenter, various elements in the experimental setup and subjects in order to achieve some aim. This characteristic is best served by computation models founded on communication, such as Milner's vision of computation as the interaction of communicating parts [28] or the tuple spaces of Linda [13].

Nondeterminism (2, 3 in Figure 1). Nondeterministic specifications and programs allow for the exploration of multiple scenarios without worrying about how each scenario is chosen, and also permit the succinct expression of various possibilities, which is convenient for the experimenter. Nondeterminism can be easily represented in process algebras such as π-calculus [29] and CSP [19], as well as in related languages (e.g., Occam [27] and, more recently, Go [33]) and libraries (e.g., JCSP [26]).

Reproducibility (1, 2, 3 in Figure 1). The possibility of reproducing experiments is fundamental in science. The fact that software is precise makes it particularly suitable for reproducibility. Interestingly, such capabilities go beyond Computer Science research through notebook-based tools [38] (see *Laboratory Environment* below).

Traceability (1, 2, 3b, 4 in Figure 1). It is often not enough to know that a certain event took place, as one might need to know

precisely which factors influenced it. For instance, causal graphs, when available, provide such traceability [32]. As another example, consider that implementations of CSP specifications using JCSP [26] can be instrumented to monitor the behaviour of the system with respect to specification elements.

Durability (3b, 4 in Figure 1). Results of experiments are valuable and therefore should be made durable. These include not only the final outcome, but also all the parameters and runtime conditions that led to it. Such information can be examined by experimenters and also used by the execution engine itself to improve future experiments.

Fault-Tolerance (2, 3b in Figure 1). Like any software process, experiments might crash in the course of their execution. Because they can be expensive to run, last for a long time and involve many subjects before any final state is reached, provisions must be taken to make EOC systems *fault-tolerant*. Moreover, partial results should be preserved whenever the nature of the experiment permits (e.g., "subjects performed with a certain efficiency before the experiment crashed").

Laboratory Environment (1, 4, 5 in Figure 1). Experimenters are central in EOC, therefore software to support their work is important, particularly with respect to user interfaces. Traditional Integrated Development Environments (IDEs) include such laboratory in the form of special tools to support test specification, execution and reporting. In Machine Learning and related areas, notebook-based interfaces such as Jupyter [23][2] and Apache Zeppelin[3] provide a convenient way to explain the problem under consideration, specify how to address it and store the results. Other examples include WEKA [16] for Data Mining, JTorx [5] for Model-Based Testing and UPPAAL [4] for Formal Verification.

3.1 The Role of Experimenters

The rise of the Data Scientist [9] shows the increasing importance of people capable of extracting knowledge from the "natural experiments" found in business data. Software developers are also being deeply influenced by such experimental concerns, both in software companies such as Microsoft [22] and within the Information Technology departments of companies in other fields (e.g., through so-called DevOps [21]). Thus, experimentation is increasingly orthogonal to other activities, and those proficient in this are critical.

EOC pushes this evolution further through the central role given to an *experimenter*, who: decides what is to be investigated or achieved; defines how data is to be generated and collected; evaluates results; and decides how to iterate. That is to say, the experimenter should not consume data only passively, as the terms "data science" and "data mining" suggest. Rather, he or she should *actively* create ways in which to interact with the system of interest, and this might mean defining or changing business functionality.

4 CONCLUSION

This text has shown through numerous examples that experimental aspects can be found dispersed and intertwined in many software systems. Other instances can be easily given. On this basis, it is proposed that experimentation should be considered a class of computation in its own right. In this manner, it is possible to abstract a general software architecture (Figure 1) and a number of supporting principles, which establish the foundation of an *Experiment-Oriented Computing* (EOC) approach to the engineering of software. Many directions can be pursued, both in building industrial EOC technologies and in advancing related research.

With respect to technologies, software libraries can be developed in order to make the software architecture and experimentation capabilities seen here available to developers in a convenient manner. These could include mechanisms to, for instance: handle various kinds of interactions with subjects; explicitly specify nondeterministic choices; manage subjects automatically; store the results of experiments and the configurations that led to them; and define formulas or other artefacts to guide program execution. Existing software could perhaps be instrumented with some of these capabilities (e.g., subjects management, to allow existing software to be easily tested with different subjects). Middleware specifically designed to handle large and distributed experimentation could be created. Tools to simplify the specification and execution of experiments could be developed, not unlike tools for Software Testing that exist today. One can also imagine Experiment-Oriented Programming languages as more concise and stricter alternatives to libraries.

In relation to research, many theoretical aspects can be examined, including: experiment evaluation (e.g., the meaning of *completeness* and *soundness*); incorporation of advanced experimental methods used in scientific practice (e.g., factorial experiments [8]); and simplification of the architecture and principles proposed here. Furthermore, it is interesting to consider what EOC can bring back to classical views on experimentation. For instance, Radder [35] reviews the Philosophical tradition and recommends further research concerning "computer experiments."

Experimentation methodology also presents practical challenges. For instance, it is unclear to what extent business stakeholders would be willing to permit experimenters to interfere with business functionality to allow for better analyses. Moreover, ethical considerations must be taken into account when subjects are humans.

As concrete examples, one can imagine many existing techniques and systems coming together under EOC, such as: profiling frameworks [39] guided by Model-Based Testing techniques [44]; user behavior analysis through scientific discovery methods [25] instead of A/B testing [24]; the infusion of simulation-based analysis [37] and Model Checking [2] in traditional Computer-Aided Design software (e.g., AutoCAD) to account for dynamic properties of designs; and Search-Based Software Engineering [18] managed in a large-scale, distributed, fashion using black-box optimization platforms [14]. The reader is invited to consider other similar combinations (perhaps by filling in the blanks in Table 1) as well as entirely novel applications of the principles described here.

ACKNOWLEDGMENTS

The author would like to thank the anonymous reviewers (including those of previous drafts) for their comments.

[2]Formerly called IPython. http://jupyter.org/
[3]https://zeppelin.apache.org/

REFERENCES

[1] Paul Ammann and Jeff Offutt. 2008. *Introduction to software testing.* Cambridge University Press.

[2] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking.* The MIT Press.

[3] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos engineering. *IEEE Software* 33, 3 (2016), 35–41.

[4] Gerd Behrmann, Alexandre David, Kim G. Larsen, John Hakansson, Paul Petterson, Wang Yi, and Martijn Hendriks. 2006. UPPAAL 4.0. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems (QEST '06).* IEEE Computer Society, Washington, DC, USA, 125–126.

[5] Axel Belinfante. 2010. JTorX: A tool for on-line model-driven test derivation and execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 266–270.

[6] Tibor Bosse, Catholijn M Jonker, Lourens Van Der Meij, and Jan Treur. 2007. A language and environment for analysis of dynamics by simulation. *International Journal on Artificial Intelligence Tools* 16, 03 (2007), 435–464.

[7] Feng Chen and Grigore Roşu. 2007. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07).* ACM press, 569–588.

[8] Linda M Collins, John J Dziak, Kari C Kugler, and Jessica B Trail. 2014. Factorial experiments. *American journal of preventive medicine* 47, 4 (2014), 498–504.

[9] Thomas H Davenport and DJ Patil. 2012. Data scientist: The Sexiest Job of the 21st Century. *Harvard business review* 90, 10 (2012), 70–76.

[10] Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lassaigne, Johan Oudinet, and Sylvain Peyronnet. 2012. Coverage-biased random exploration of large models and application to testing. *International Journal on Software Tools for Technology Transfer* 14, 1 (2012), 73–93.

[11] Sašo Džeroski and Ljupčo Todorovski (Eds.). 2007. *Computational Discovery of Scientific Knowledge.* Lecture Notes in Computer Science, Vol. 4660. Springer Berlin Heidelberg.

[12] Krzysztof Gajos and Daniel S Weld. 2004. SUPPLE: automatically generating user interfaces. In *Proceedings of the 9th international conference on Intelligent user interfaces.* ACM, 93–100.

[13] David Gelernter. 1985. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7, 1 (1985), 80–112.

[14] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. 2017. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM, 1487–1495.

[15] Ronald Apriliyanto Halim and Mamadou Diouf Seck. 2011. The Simulation-based Multi-objective Evolutionary Optimization (SIMEON) Framework. In *Proceedings of the 2011 Spring Simulation Multiconference (SpringSim'11).* The Society for Modeling and Simulation International.

[16] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

[17] Per Brinch Hansen. 2001. The invention of concurrent programming. In *The origin of concurrent programming.* Springer, 3–61.

[18] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and Software Technology* 43, 14 (2001), 833–839.

[19] Charles Antony Richard Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (1978), 666–677.

[20] Claude Jard and Thierry Jéron. 2005. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.* 7, 4 (2005), 297–315.

[21] Gene Kim, Patrick Debois, John Willis, and Jez Humble. 2016. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations.* IT Revolution.

[22] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2016. The emerging role of data scientists on software development teams. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on.* IEEE, 96–107.

[23] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter Notebooks-a publishing format for reproducible computational workflows.. In *ELPUB.* 87–90.

[24] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M Henne. 2009. Controlled experiments on the web: survey and practical guide. *Data mining and knowledge discovery* 18, 1 (2009), 140–181.

[25] Pat Langley, Herbert A. Simon, Gary L. Bradshaw, and Jan M. Zytkow. 1987. *Scientific discovery: Computational explorations of the creative processes.* MIT press.

[26] Doug Lea. 1999. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley Longman Publishing Co., Inc. See also `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[27] INMOS Limited. 1984. *Occam programming manual.* Prentice Hall Direct.

[28] Robin Milner. 1993. Elements of Interaction: Turing Award Lecture. *Commun. ACM* 36, 1 (Jan. 1993), 78–89.

[29] Robin Milner. 1999. *Communicating and mobile systems: the pi calculus.* Cambridge University Press.

[30] Tom Mitchell. 1997. *Machine learning.* McGraw Hill.

[31] Jeffrey Nichols, Duen Horng Chau, and Brad A Myers. 2007. Demonstrating the viability of automatically generated user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems.* ACM, 1283–1292.

[32] Judea Pearl. 2009. *Causality.* Cambridge University Press.

[33] Rob Pike. 2012. Go at Google. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity.* ACM, 5–6.

[34] Juan C Quiroz, Sushil J Louis, Anil Shankar, and Sergiu M Dascalu. 2007. Interactive genetic algorithms for user interface design. In *IEEE Congress on Evolutionary Computation (CEC 2007).* IEEE, 1366–1373.

[35] Hans Radder. 2009. The philosophy of scientific experimentation: a review. *Automated Experimentation* 1, 1 (2009), 2.

[36] Paulo Salem. 2017. User Interface Optimization Using Genetic Programming with an Application to Landing Pages. *Proc. ACM Hum.-Comput. Interact.* 1, 1 (June 2017), 13:1–13:17.

[37] Paulo Salem and Ana CV De Melo. 2013. On-the-fly verification of discrete event simulations by means of simulation purposes: Extended version. *Simulation* 89, 8 (2013), 977–1008.

[38] Helen Shen. 2014. Interactive notebooks: Sharing the code. *Nature News* 515, 7525 (2014), 151.

[39] Marcio Silva, Michael R Hines, Diego Gallo, Qi Liu, Kyung Dong Ryu, and Dilma da Silva. 2013. CloudBench: Experiment Automation for Cloud Environments. In *Cloud Engineering (IC2E), 2013 IEEE International Conference on.* IEEE, 302–311.

[40] Dag IK Sjøberg, Jo Erskine Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, N-K Liborg, and Anette C Rekdal. 2005. A survey of controlled experiments in software engineering. *IEEE transactions on software engineering* 31, 9 (2005), 733–753.

[41] Andrew Sparkes, Wayne Aubrey, Emma Byrne, Amanda Clare, Muhammed N Khan, Maria Liakata, Magdalena Markham, Jem Rowland, Larisa N Soldatova, Kenneth E Whelan, et al. 2010. Towards Robot Scientists for autonomous scientific discovery. *Automated Experimentation* 2, 1 (2010), 1.

[42] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction.* Vol. 1. MIT press.

[43] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 847–855.

[44] Jan Tretmans. 2008. Model Based Testing with Labelled Transition Systems. In *Formal Methods and Testing (Lecture Notes in Computer Science)*, Robert M. Hierons, Jonathan P. Bowen, and Mark Harman (Eds.), Vol. 4949. Springer, 1–38.

[45] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann.