Collecting Usage Data for Software Development: Selection Framework for Technological Approaches

Sampo Suonsyrjä, Kari Systä, Tommi Mikkonen and Henri Terho
Tampere University of Technology, Korkeakoulunkatu 1, FI-33720 Tampere, Finland
{sampo.suonsyrja, kari.systa, tommi.mikkonen, henri.terho}@tut.fi

***Abstract*—Software development methods are shifting towards faster deployments and closer to the end users. Their ever tighter engagement of end-users also requires new technologies for gathering feedback from those users. At the same time, widespread Internet connectivity of different application environments is enabling the collection of this post-deployment data also from sources other than traditional web and mobile software. However, the sheer number of different alternatives of collecting technologies makes the selection a complicated process in itself. In this paper, we describe the process of data-driven software development and study the challenges organizations face when they want to start guiding their development towards it. From these challenges, we extract evaluation criteria for technological approaches to usage data collecting. We list such approaches and evaluate them using the extracted criteria. Using a design science approach, we refine the evaluation criteria to a selection framework that can help practitioners in finding a suitable technological approach for automated collecting of usage data.**

I. Introduction

One of the clear trends in the field of software development has been the ever tighter engagement of the end-users to the software development process. For example, methods such as Lean Startup [1] are dependent on more and more rapid feedback cycles. As described in a more general level in [2], the shift from Agile processes towards Continuous Deployment and experiment systems requires faster ways to validate the developed software than is possible with traditional communication methods, such as face to face conversations with end-users.

As these new methods are emerging, the whole software development process can be rearranged. In the aforementioned experiment systems for example, the deployment of software is not the end of the road for development efforts, but more of an initial step to start collecting data on user needs and then fine-tune the software [2]. With such approach, post-deployment data is first collected and then used for guiding the software development making the development process data-driven.

First and foremost this post-deployment data, such as data about how the system is used (i.e. usage data), has been used for guiding software development in environments like web and mobile development. In these contexts, constant connectivity – an important enabler for usage data collection – is the norm. However, breakthroughs of cloud software and Software-as-a-Service model, and the fact that most applications and platforms are Internet connected to begin with, are extending the use of data collection to a wider range of applications.

DOI reference number: 10.18293/SEKE2016-186

As this range of potential target applications, or programs whose usage data can be collected from, is getting wider, we are left with the challenge of finding the right technological approach for usage data collecting in the varying target application environments and cases. For example, manually adding code to target applications for logging purposes can be a straightforward option for developers in simple cases on one hand. But on the other hand, there are also different kinds of standardized tools and various approaches that among other things can automate this instrumentation or at least some parts of it.

To address this, we study what kind of challenges organizations face when they are starting the usage data collecting. Literature reviews along with a case study in an international telecommunication organization are used for finding these challenges, and they are extracted into evaluation criteria for data collecting technologies. We then describe several options for the automatic usage data collecting and evaluate them with the formed criteria. After this, we refine the criteria and the evaluated technological approaches into a selection framework that should help practitioners choose the suitable technologies.

The main research problem is *how to select the right technological approach for automated collecting of usage data?*. To address this, we derive two research questions from the main problem as follows.

- RQ1: How to evaluate different technological approaches for automated collecting of usage data?
- RQ2: What kind of technological approaches are there for the automated collecting of usage data?

The rest of the paper is structured as follows. In Section II, we take a look at the context of data-driven software development. Additionally, we go through the appropriate literature to find out challenges in automatic collecting of post-deployment data. Section III explains the formed evaluation criteria, and in Section IV we use the criteria to evaluate several technological approaches to usage data collecting. In Section V we derive a selection framework from the evaluation criteria and the technological approaches. In Section VI we draw some final conclusions.

II. Background

The background of this paper is two-fold. First, we address data-driven software development. Then, we introduce the

challenges of automatic usage data collecting.

A. Data-Driven Software Development

As presented in [2], companies typically evolve their software development processes by climbing the *Stairway to Heaven* (StH). StH describes the shift from traditional waterfall development towards continuous deployment of software. The steps to be taken in the proposed chronological order are *Traditional Development, Agile R&D Organization, Continuous Integration, Continuous Deployment*, and the model ends up with *R&D as an Experiment System*. With each step, software development is becoming faster in the sense that it produces new releases of software ever more quickly. In the scope of this paper, the last phase is especially interesting as climbing the last step requires a fast-track of information from customers back to the development organization.

However, feedback gathering from customers is often slow, and sufficient mechanisms for it are missing. This can result in opinion-based development decisions. To ease the climb to the final step and make development more data-driven, Olsson & Bosch have developed the HYPEX model, i.e. *Hypothesis Experiment Data-Driven Development* [3]. In this model, *Minimal Viable Features* (MVF) are implemented and their expected behavior is described. A feature is implemented over many iterations and so the first 10% to 20% of its functionality is called an MVF. The deployed MVF is always instrumented to collect data on its use by customers, and this data is then compared with the initial descriptions of how the development organization thought that it would be used. Based on this *Gap Analysis*, the developers then either finalize or abandon the feature, or iterate the experimentation over again with a different hypothesis.

Such process has a lot in common with the *Build-Measure-Learn loop* (BML-loop) described in [1].Compared to the HYPEX model, the BML loop has many similarities and main differences are in the abstraction level. The BML loop is meant to validate the business feasibility of the product through the use of *Minimum Viable Products* (MVP). During each turn of the BML-loop a product hypothesis is formed and measurable metrics are linked to the hypothesis. The MVP is then built and the metrics are measured. Based on the outcomes of this data, the decision is made if the product development should be continued or another product hypothesis should be tested based on the experiences learned from the MVP.

As described above, both the HYPEX model and the BML-loop are data-driven approaches to software development. *Software Analytics*, as laid out in [4], highlights this use of data as well. However, this paradigm of analytics points out specifically the different types of analyses that are needed for turning the measured data into insights and eventually into development decisions. These are depicted in Figure 1. The model originates from the field of web analytics, but as its generality seems broad and with its experimental approach it should suit organizations well as a guidance in the *R&D as an Experiment System* phase of StH.

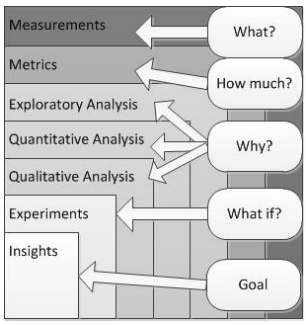

Fig. 1. Paradigm of Analytics (adapted from [4]).

All the aforementioned models include the phases of planning the data collecting, collecting the data, and analyzing the results to make decisions and iterating the process over again. In this sense, data-driven software development can be seen as an overarching term that typically consists of similar phases. To get a concrete definition from a technology standpoint and in the scope of this paper, we have formulated *Data-Driven Software Development* as an iterative process as follows.

1) *Planning of the data collection.* The goals of the analysis need to be known and the monitored applications and features should be selected based on them. The required resources, customer and user permissions and legal aspects of data collection need to be checked as well.
2) *Deployment of data collection.* The infrastructure of technical means to track the applications and collect post-deployment data needs to be installed.
3) *Monitoring of the applications.* The technical means can be internal to the application but also external - depending on the used run-time and platform technologies.
4) *Picking up the relevant data.* Monitoring should be configured to pick the data that is seen useful for the planned data collection and analysis.
5) *Pre-processing – filtering and formatting – the data.* The collected data is typically transferred to a remote location, but is typically filtered and formatted before sending to save resources.
6) *Sending and/or saving the data*. For effective analysis the data needs to be collected from long enough period and it needs to be available for the people working on the analysis. Often this means that hosting of the data storage is different from the applications. Thus, the system should transfer the data to storage either by means of continuous streaming or by saving it first to local cache and sending bigger amounts of data at the same time.
7) *Cleaning and unification of the data.* This process completes the work done by pre-processing described earlier but is necessary especially if data is flowing from various different sources.
8) *Storing the data.* Typically some database is used for storing the data.
9) *Visualizations and analysis.* A tools set helps stakehold-

ers to ask "what" and "how much" questions and to make conclusions.

10) *Decision making.* The results should lead to actionable decision for example on: new software development, user training, or marketing actions.

In this process, data collecting consists of phases 2-6.

B. Challenges of Automatic Usage Data Collecting

Fabijan et al. have described the challenges and limitations of customer feedback and data collection techniques in their literature review of software R&D [5]. The scope of their literature review included also manual and qualitative techniques such as interviews and observations, but the sources and challenges concerning automatic usage data collecting from the software product itself were as listed below.

- *Incident reports*: Available only after an incident.
- *Beta testing*: Only partially developed interfaces and functionality.
- *Operational and event data*: Security issues when such data is transmitted, potentially high amounts of data.
- *A/B testing*: Potentially confusing for customers when exposed to different versions.

Similarly, Sauvola et al. [6] described feedback gathering and its challenges as a part of software development companies' R&D efforts. Although their multiple-case study involved also many more feedback types than the automatically collected usage data, their descriptions of the cases implied various related challenges. We understood these as follows.

- *Permission checks.* The authors point out that in some specific domains the automated data collection from end-users is highly regulated and thus not executed at all.
- *Various sources of feedback.* Consolidating the feedback coming from various customers was seen as a challenge, and its processing relied heavily on its user's competence.
- *Only incident reports available.* Feedback was only gathered for troubleshooting purposes and not e.g. for improving existing products.
- *Systematic implementations are missing.* Although some mechanisms are in place to collect feedback and even product data, their implementations lack the systematic approach.
- *Difficulties to store, analyze, and integrate.* Even if feedback was gathered in most of the case companies, they reported having issues in storing, analyzing and integrating it back to the developers' processes.
- *Data availability and transparency.* The information about the collected data types as well as who and for what was it used was difficult to spread around the case companies. A reason for this, for example, was that the different parts of the development organization can see the data collecting as a risk as its use for new product development can cannibalize the current product markets.
- *Channels are not working.* As no systematic and organization-wide ways of feedback collecting were present, the feedback gathered in one place was regarded useless although it could have been in high value in the next place.

III. Evaluation Criteria for Usage Data Collecting Approaches

The challenges of usage data collecting are now extracted into evaluation criteria, which are fine-tuned based on the discussions with the case company. The challenges and limitations of usage data collecting can be consolidated as follows:

- *The amount of use cases for the collected data.* The number of use cases can be either too low or too high. Although there could be various uses for the same collected data, it might be used blindly to serve only a single purpose. On the other hand, the whole data collecting can face the critical challenge of trying to serve so many purposes and people that in the end it performs sufficiently to none of them.
- *The timeliness of the collected data.* Depending on the intended use, the timeliness of the data can form limitations to the collecting approach as well as to the source of data. For example, incident reports can be available only after incidents happen, and thus the use of such source has its natural challenges.
- *Continuous confusion for the users.* As mentioned, one of the well-known practices in the field of web development is A/B testing. Its implementation needs carefully planning, though. The more continuous the collecting is, the higher the risk of continuously introducing partially developed interfaces and functionality to users, who can find this troubling after a while. In addition, the collecting can affect the performance of the system.
- *Laws, regulations, and permissions.* Especially when the same data collecting approach is to be used in various different domains and countries, the related laws and regulations are going to be different for each situation. These checks for the data collection's legality take different amounts of time in each case thus enabling the data collecting in different cases at different times.
- *Privacy and security.* In addition to the overall permission checks, the security and privacy issues need to be addressed sufficiently by the collecting approach.
- *Various sources for the data collecting.* A high amount of sources creates a twofold challenge. The unification of the different types of data has to happen in a phase of its own (cf. phase 7 in Section II-A), or then the analyzer (cf. phase 9) has to have the capabilities to present and process the different types of data.
- *Lack of a systematic approach to collecting.* If a systematic approach is missing for the collecting, it is obvious that each phase of the data-driven software development is going to present new difficulties and challenges (e.g. difficulties to store, analyze, integrate etc.). These might be different in each case and they depend on the involved persons and their capabilities.
- *Availability, transparency, and usability of the collected data.* Even if the organization had first decided on what

kind of things they want to use the collected data, there are challenges also in how to make the data available, attractive, and usable for the right people. Thus, both the channels for distributing but also the tools for example for visualizing it (i.e. make the data usable) have to be sufficient enough for the selected audience.

We also analyzed these challenges from the perspective of the case company and used them as the basis of designing the evaluation criteria of usage data collecting approaches. The organization listed the challenges they felt were related to their case after discussing the overall topic of usage data collecting with us. Although each of the challenges they listed was found already from the list above, their descriptions of the challenges bring understanding to a more concrete level, which we try to emphasize with the examples linked to each criterion.

- *Timeliness.* When can the data be available? Does it have a support for real-time?
- *Targets.* Who should benefit from the data? What is the intended use? Does the approach support many targets? Does it produce different types of data or only one? "Do we want results for troubleshooting or for new product development?"
- *Effort level.* What kind of a work effort is needed from the developers to implement the approach. "How does the selected technology affect the production code? What is the work effort needed for the implementation?"
- *Overhead.* What kind of drawbacks are acceptable? "How does the collecting approach affect the implementation environment, e.g. downtime and performance?"
- *Sources.* What sources of data can be used? Does the approach support many source platforms? "Different kinds of technological environments – Where to focus our implementation efforts?"
- *Configurability.* How configurable the technological approach is? Can the collecting be switched on and off easily? Can it change between different types of data to collect? "Is the collecting easy to switch on and off? Is the approach producing data in the right level of details?"
- *Security.* Can the organization who developed the collecting technology be trusted with the collected data? Is the data automatically stored by the same organization?
- *Reuse.* How can the technology be reused? Is it always a one-time solution or can it be reused as it is straightforwardly with another target application?

IV. Technological Approaches for Usage Data Collection

Next, we will go through a few technological approaches for the automated collecting of usage data. The abstract viewpoint is selected to not get stuck with the specific tools that happen to be around in 2016. Rather, the goal is to gain deeper understanding in how such tools and possible approaches work and how that is reflected in selecting them.

A. Manual Implementation

In the manual implementation the developer adds extra statements to the relevant locations of the software. On one hand, this highlights the flexibility of the approach – it does not limit the *timeliness, targets, sources,* or *security* in any way. On the other hand, adoption to new targets and sources would require significant rework making the *reuse* practically impossible. However, if additional functionalities such as runtime flags are added to the statements, switching the collecting on and off becomes significantly easier. This increasing *configurability* correspondingly increases the already high level of work *effort* needed for the implementation though. As a benefit, the approach almost guides the developer to collect data only from the intended sources, minimizing the *overhead* to the performance and of irrelevant data.

To conclude, there are only few real limitations with this approach. The needed work effort is high though, so e.g. if the code base is vast the approach can be come inappropriate. Therefore, we conclude that the approach is best suited either for the first few try outs with data collecting or for cases where the target and the source are particularly well focused.

B. Automatic Instrumenting with a Separate Tool

There are multiple tools, e.g. GEMS [7], that can automatically instrument the code for various data logging, quality assurance and performance monitoring purposes. This approach frees the programmers from the manual work and reduces the probability for errors lowering the *effort* significantly. Similarly, the *reuse* possibilities of the data collection should be high with automated tools since they are developed to work with different target applications in the first place.

However, the automatic tools are typically focused on only one type of *source* or *target*. Therefore, the *overhead* is likely to grow rapidly as the source cannot be set as specifically as with the manual approach. There are exceptions to this as well, and for example the framework presented in [8] balances its monitoring coverage with overhead automatically. Although these problems in general can be reduced by using highly *configurable* instrumentation tools when available, these criteria, along with *security and timeliness*, are almost completely intertwined with the specific tool selected. This highlights the inflexibility of the technological approach. The ideal case for this approach could be one with high importance in low implementation effort, such as a case with a huge code base, and with targets that need monitoring from the whole target application or even from many similarly developed target applications.

C. Aspect-Oriented Approach

Aspect-oriented approach is something of a mixture from the automatic instrumentation and the manual implementation. The research presented in [9] and [10] use aspect-oriented programming as a tool for code instrumentation. Further on, the use of aspect-oriented programming for usage data collection has been proposed in [11]. Additionally, in [12] the separation of similar monitoring code from the actual

system code is highlighted, which could perhaps respond to the challenge of various data sources.

One important benefit of aspect-orientation is its expressive power. While automatic instrumentation is typically triggered by entering (or leaving) a function, the aspects can include more complex conditions for executing the data collection code. Aspect-based instrumentation allows the instrumentation to be system and application specific, which focus the collecting better on the relevant *targets*. This should also lead to optimized balance between the additional *overhead* and quality of the data.

The expressive power of AOP makes the approach similar to the manual implementation in its flexibility to create solutions that can be optimized by their *timeliness, configurability, and security* to suit any situation. On the other hand, the work *effort* needed for the implementation is not as high since the instrumentation is automated. However, learning to use AOP surely takes its toll if the developer is not familiar with the paradigm otherwise.

From the perspective of *reuse*, the aspect-oriented approach has both its limitations as well as benefits. If the different target applications are developed in such a similar fashion that the targeted data collecting places use the same syntax, the reuse should be very straightforward. Obviously, this sets a strict limitation to the reuse. On a more general level, the approach is depended on an available AOP library for the specific target application's programming language. If the language changes between the *sources*, i.e. the target applications, the reuse becomes much more difficult. This approach suits particularly well cases which need the same kind of system wide monitoring as the tool instrumentation's ideal case, but which at the same time require more flexibility from the collecting. An available AOP library for the case's programming language is obviously a critical limitation.

D. Alternative Implementation of a UI Library

An alternative implementation of a user-interface (UI) library can be set to automatically collect usage data. Because the user interaction is usually implemented with standard UI libraries, their components can be altered so that they include the collection of usage data within them. Similarly to automatic instrumentation, this approach frees the developers from the repetitive implementation *efforts*. Correspondingly, the issues are similar as well – data is easily collected also from unnecessary *sources* causing extra performance *overhead* and difficulties to the analysis phase. Although usage data is mainly linked to the UI and the types of data a UI library includes, some *targets* need integration with data types that are beyond the reach from these altered UI libraries.

On a more positive note, the approach has no limitations to the *security* or *timeliness*, and the *configurability* can be increased much like in the manual approach. As a matter of fact, this can be even easier if a differently altered UI library is deployed according to each requirements of a new case. The *reuse* of the implementations with this approach can be extremely easy, but new versions of the standard UI libraries create great issues as well.

TABLE I
SUMMARY OF THE TECHNOLOGICAL APPROACH EVALUATIONS.

	Technologies				
Criteria	*Man.ins.*	*Tool ins.*	*AOP*	*UI*	*E.E.*
Timeliness	+	-	+	+	-
Targets	+	-	+	-	-
Effort	-	+	+	+	+
Overhead	+	-	+	-	-
Sources	+	-	-	-	-
Config.	+	-	+	+	-
Security	+	-	+	+	-
Reuse	-	+	-	-	+

E. Execution Environment

The data collection can also be done by the environment without any modification to the application. For languages like Java and JavaScript the virtual machine is an execution environment where method and function calls can be monitored by instrumenting critical places. One example of such systems is Patina [13] where the user input like cursor movements and key-presses are monitored. In these approaches, the implementation *effort* is often low, but the produced data requires a lot of post processing and there may significant performance penalties since it will reduce possibilities for advanced just-in-time compilation.

This approach often has a limited set *sources* and *targets*, but within that limited scope *reuse* is good. Similar to the automatic instrumentation tools, the *configurability, security*, and *timeliness* of the approach are intertwined heavily with the specific implementation, i.e. tool, that is implemented.

V. SELECTION FRAMEWORK FOR AUTOMATED USAGE DATA COLLECTING TECHNOLOGIES

We have summarized the evaluations of the technological approaches into the basis of the selection framework, i.e. Table I, giving each approach either a plus if it has a positive impact or if it does not restrict the implementation. An approach is marked with a minus sign if it limits the selection or the use of a data collecting implementation according to each criteria.

The first thing to do when selecting a technological approach to usage data collecting, is to rapidly **explore the case** to get a grasp of the most critical limitations to the technological approaches. These include things such as the size of the code base, availability of automated tools and AOP libraries for the target application's language and platform, and access to the UI libraries and execution environments.

If any critical limitations are faced, the next step is to **reject the unsuitable approaches** accordingly. For example, if there are many security issues related to the data being collected or if data needs to be sent in real-time, the 3rd party tools used in approaches B and E might have critical limitations that cannot be avoided.

The following step is to **prioritize the evaluation criteria**. In addition to the explored case information, one should find

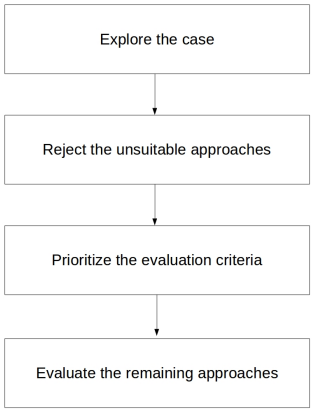

Fig. 2. Selection Framework for Technological Approaches.

out the goals different stakeholders have for the usage data collecting as these can have a major impact on the approach selection. If the goals are clearly stated, and the aim is e.g. to simply find out which of two buttons is used the most, manual instrumentation can work sufficiently. However, if the goal is stated anything like "to get an overall view of how the system is used" or if the goal is not stated at all, the more automated and more configurable approaches most likely become more appealing. Therefore, one of the most crucial things to find out in this step is to understand what different stakeholders want to accomplish with the collected data.

After this, the final step is to **evaluate the remaining approaches**. The plus and minus signs used in Table I work as guidelines in this, but their emphasis obviously varies on a case to case basis. To summarize, the selection framework is illustrated in Figure 2.

The further evaluation of how the selection framework performs is clear choice for future work. The setting with the case company is interesting for them, but it is attractive also academically as it provides an environment to study the "full-stack" that will be needed for the whole data-driven software development process in the end.

VI. Conclusions

In this paper, the main research problem was *how to select the right technological approach for automated collecting of usage data*. Literature reviews were performed to gain understanding of the context of the collecting processes and its challenges. These helped us form evaluation criteria for the technological approaches. We then described different approaches and evaluated them with the aforementioned criteria, which were then refined into the selection framework.

To summarize, the main contributions of this paper included literature reviews of the data-driven software development and of the challenges of collecting usage data, forming the evaluation criteria based on the studied challenges, description and evaluation of different technological approaches, and designing the selection framework for the technological approaches. The selecting of data collecting technologies is not a straightforward challenge but it needs to be addressed each time an organization wants to start the data-driven software development. Obviously, various contemporary tool evaluations are available both in academic literature and especially in practitioner publications, e.g. blogs. However, the abstract perspective of this study to the technological approaches rather than today's tools should be valuable also in the long run.

Finally, the criteria described in this paper gives practitioners a good basis for the evaluation, but it works only as a guideline and case specific variations account heavily on the actual decisions. However, it provides them with a well-considered starting point in their journey towards ever more data-driven decision making.

Acknowledgment

This work is supported by Tekes (http://www.tekes.fi/) and Digile's Need for Speed program (http://www.n4s.fi/en/).

References

[1] E. Ries, *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*. Crown Books, 2011.

[2] H. H. Olsson, H. Alahyari, and J. Bosch, "Climbing the" stairway to heaven"–a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software," in *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. IEEE, 2012, pp. 392–399.

[3] H. H. Olsson and J. Bosch, "From opinions to data-driven software r&d: A multi-case study on how to close the 'open loop' problem," in *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*. IEEE, 2014, pp. 9–16.

[4] R. P. Buse and T. Zimmermann, "Information needs for software development analytics," in *Proceedings of the 34th international conference on software engineering*. IEEE Press, 2012, pp. 987–996.

[5] A. Fabijan, H. H. Olsson, and J. Bosch, "Customer feedback and data collection techniques in software r&d: a literature review," in *Software Business*. Springer, 2015, pp. 139–153.

[6] T. Sauvola, L. E. Lwakatare, T. Karvonen, P. Kuvaja, H. H. Olsson, J. Bosch, and M. Oivo, "Towards customer-centric software development: A multiple-case study," in *Software Engineering and Advanced Applications (SEAA), 2015 41st Euromicro Conference on*. IEEE, 2015, pp. 9–17.

[7] P. K. Chittimalli and V. Shah, "GEMS: A Generic Model Based Source Code Instrumentation Framework," in *Proceedings of the Fifth IEEE International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2012, pp. 909–914.

[8] J. Ehlers and W. Hasselbring, "A self-adaptive monitoring framework for component-based software systems," in *Software Architecture*. Springer, 2011, pp. 278–286.

[9] W. Chen, A. Wassyng, and T. Maibaum, "Combining static and dynamic impact analysis for large-scale enterprise systems," in *Product-Focused Software Process Improvement*. Springer, 2014, pp. 224–238.

[10] A. Chawla and A. Orso, "A generic instrumentation framework for collecting dynamic information," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, pp. 1–4, Sep. 2004. [Online]. Available: http://doi.acm.org/10.1145/1022494.1022533

[11] S. Suonsyrjä and T. Mikkonen, "Designing an unobtrusive analytics framework for monitoring java applications," in *Software Measurement*. Springer, 2015, pp. 160–175.

[12] M. Vierhauser, R. Rabiser, P. Grünbacher, K. Seyerlehner, S. Wallner, and H. Zeisel, "Reminds: A flexible runtime monitoring framework for systems of systems," *Journal of Systems and Software*, vol. 112, pp. 123–136, 2016.

[13] J. Matejka, T. Grossman, and G. Fitzmaurice, "Patina: Dynamic heatmaps for visualizing application usage," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '13. New York, NY, USA: ACM, 2013, pp. 3227–3236. [Online]. Available: http://doi.acm.org/10.1145/2470654.2466442