**Paper # 1:**

**From Ad-Hoc Data Analytics to DataOps:**

Aiswarya Raj, Helena, and Anas Dakkak (2020) empirically invested Ad-Hoc Data Analytics to DataOps. Accepted for the International Conference on Software and Systems Process (ICSSP 2020) June 26-28, 2020, Seoul, South Korea.

**INTRODUCTION:**

The collection of high-quality data provides a key competitive advantage to companies in their decision-making process. It helps to understand customer behavior and enables the usage and deployment of new technologies based on machine learning. However, the process from collecting the data, to clean and process it to be used by data scientists and applications is often manual, non-optimized and error-prone. This increases the time that the data takes to deliver value for the business. To reduce this time companies are looking into automation and validation of the data processes. Data processes are the operational side of data analytic workflow. DataOps, a recently coined term by data scientists, data analysts and data engineers refer to a general process aimed to shorten the end-to-end data analytic life-cycle time by introducing automation in the data collection, validation, and verification process. Despite its increasing popularity among practitioners, research on this topic has been limited and does not provide a clear definition for the term or how a data analytic process evolves from ad-hoc data collection to fully automated data analytics as envisioned by DataOps. This research provides three main contributions. First, utilizing multi-vocal literature we provide a definition and a scope for the general process referred to as DataOps. Second, based on a case study with a large mobile telecommunication organization, we analyze how multiple data analytic teams evolve their infrastructure and processes towards DataOps. Also, we provide a stairway showing the different stages of the evolution process. With this evolution model, companies can identify the stage which they belong to and also, can try to move to the next stage by overcoming the challenges they encounter in the current stage. Data is the key asset for the organizations as it helps in better decision making, analyze performance and solving problems, to analyze the consumer behavior and market and so on. Moreover, data is the backbone for many hot and trending technologies like machine learning and deep learning. Increased importance of data lead to the acquisition and storage of data in higher volumes which in turn gave rise to fields like Big Data, data mining and data warehousing. Due to the operations initiated by the engineers or by the change in data sources, continuous data changes happen and there is the requirement for data versioning and sharing techniques.

**METHODOLY:**

The goals of this study was to formulate a definition for DataOps and to identify the phases of DataOps evolution. RQ seeks to identify data strategies that are used at Ericsson to do Data Analytics in industrial systems and also the drivers for adoption at each stage to move to the successive stages. overlap between the two data analytic approaches and the practical difficulties Ericsson encounter while trying to completely adopt DataOps as their analytic approach.To set the basic understanding on DataOps concepts and the essential components, we adopted the Multi-Vocal Literature Review approach following the instructions given by Then we conducted an interpretive single-case study, following the guidelines by to acquire a deeper understanding on the data analytic approach followed at Ericsson. The main focus of this study is to understand and explain how the DataOps approach is perceived by Data Scientists, Data Analysts and Data Engineers to shorten the end to end data analytic life-cycle time and to enable collaboration. The impediments identified are based on our interpretations of the experiences of experts who work with data in a real-time scenario with real-world data collected from edge devices. The multiple cases from different teams in the same company are used in this study because it facilitates the exploration of a particular concept in a real life setting as well as through a variety of lenses.

**FINDINGS:**

It represents a definition of DataOps derived from literature as well as from the definitions given by experts during the interview study. Based on the study, we have constructed a five stage evolution model of identified five stages of evolution of data strategy adopted at Ericsson to meet evolving requirements of the customer. Our study is carried out with ten use cases as mentioned above. These use cases are mapped to the corresponding stage in the evolution model. Also, impediments encountered at each phase and the driving factors for the next phase are identified.

**RESULT:**

DataOps is becoming increasingly popular in the industry due its ability to accelerate the production of high quality data insights. This paper proposes an evolution model with five phases in which data strategies adopted by the company at different points of time. With our research contribution, which is based on an extensive case study at Ericsson, we aim to provide guidance on this topic and enable other companies to establish or scale their DataOps practices. Our main contribution is the “DataOps Evolution Model”. In the model, we summarize the four phases of evolution and maps use cases in the study to appropriate phase of the model. Researchers and practitioners can use this model to position other case companies and guide them to the next phase by suggesting the necessary features. As future research, we plan to validate our model with other companies.

**Paper # 2:**

**An Empirical Study on the Impact of Deimplicitization on Comprehension in Programs Using Application Frameworks.**

Jürgen Cito, Jiasi Shen, and Martin Rinard (2020) study An Empirical Study on the Impact of Deimplicitization on Comprehension in Programs Using Application Frameworks In 17th International Conference on Mining Software Repositories (MSR ’20), October 5–6, 2020, Seoul, Republic of Korea. ACM, New York,

**INTRODUCTION:**

Application frameworks, such as Ruby on Rails, introduce abstractions with the goal of simplifying development for particular application domains, such as web development. While experts enjoy increased productivity due to these abstractions, the flow of the programs is often hard to understand for non-experts and newcomers due to implicit flow and concealed lower level action that seems like “magic”. They conjecture that converting these implicit flows into an explicit and unified form can help non-experts comprehend the programs using these frameworks. We call the process of unifying distributed, implicit flows into a single routine deimplicitization. To deimplicitize programs that use these frameworks, we implemented an approach, Konure [6], that uses active learning to infer programs that access relational databases. The Konure paper presents experiments where Konure infers the functionality of programs in various languages (Ruby on Rails, Java, and Python) and regenerates the functionality in Python. They want to conduct an experiment that studies the impact of deimplicitization on program comprehension. Particularly, we want to study how software developers with different expertise (novices/students, framework experts/professional developers) can answer comprehension questions differently with respect to time and correctness, under the treatments of either a deimplicitized version of the program in Python or the original version of the program in Ruby on Rails.

**METHODOLY:**

Konure executes the program, the Konure proxy collects the traffic between the program and the database. As a result Konure observes all of the low-level SQL queries performed by the program. Note that these queries are often hidden from developers — the application frameworks are designed to implicitly perform low-level queries based on high-level source code. Konure uses an internal domain specific language (DSL) to represent the observed program functionality. This DSL supports (a subset of) SQL queries, along with variable references and control structures such as conditional branches and bounded loops. Konure works with black box programs whose externally visible behavior conform to the DSL. Because Konure does not need to analyze the source code of these programs, the programs may be implemented in any languages or application frameworks. Given a potentially implicit program whose behavior conforms to the DSL, Konure infers the program functionality and translates it into a unified and explicit Python routine. When the original program is implemented with implicit application frameworks such as Ruby on Rails, the regenerated Python routine serves as a deimplicitized version of the original program.

**FINDINGS:**

To provide an overview of the study results, we plan to present summary statistics (mean, median, standard deviation, max, min) over the treatment groups and overall for the dependent variables (time in seconds, number of correctly fulfilled tasks). Additionally, we want to show the distribution as part of a histogram. As for significance tests, they plan to follow a frequentist approach. They would first test for normality by conducting a Shapiro-Wilk test. Depending on the distribution, we would then either perform a test or Mann-Whitney U test.We will do this for our variables Time Spent and Correctness. Since correctness is a binary variable, our measure will be the number of correct answers given. They initially calculated effect size as part of our power analysis. They also set out to estimate the extent of how substantially different the measures in our full dataset are by calculating the effect size again. We choose the particular procedure to calculate effect size given the distribution of our sample measures. Cohen’s d is an appropriate comparison between two means, usually to accompany a t-test. Cliff’s delta is a non-parametric effect size measure that is used to calculate the frequency of values in one distribution differ from values in another distribution.

**RESULT:**

Using a specific sample of small to medium size Ruby on Rails applications and APIs could limit the ability to generalize the results, especially for time, to larger applications or applications with different code quality (e.g., availability of comments). They anticipate that the results for correctness and expertise are still relevant for larger applications, given that the focus is on one input/output flow targeting a particular entry point of the application. Using a research tool (Konure) could limit the ability to generalize the results to deployment scenarios. We mitigate this threat by preprocessing the Konure deimplicitization outputs systematically to mimic potential deployment scenarios. Focusing on only one abstraction framework (Ruby on Rails) could limit the ability to generalize the results to other abstraction frameworks, especially frameworks that adopt different abstraction strategies. However, the particular abstractions (model abstraction, view abstraction, routing, inversion of control) that we target with Konure are traits that are prevalent in many other application frameworks, such as Python/Django, PHP/Symfony, and Java/Spring. This increases the likelihood that our results generalize within the scope of these application frameworks. Using only one deimplicitization tool could limit the ability to generalize the results to other potential tools that deimplicitize programs in other domains of computation.

**Paper # 3:**

**Testing of Mobile Applications in the Wild: A Large-Scale Empirical Study on Android Apps**

Fabiano Pecorelli, Gemma Catolino, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2018. Testing of Mobile Applications in the Wild: A Large-Scale Empirical Study on Android Apps. In ICPC 2020: IEEE/ACM International Conference on Program Comprehension, May 23–24, 2020 - Seoul,

**INTRODUCTION:**

Nowadays, mobile applications (a.k.a., apps) are used by over two billion users for every type of need, including social and emergency connectivity. Their pervasiveness in today’s world has inspired the software testing research community in devising approaches to allow developers to better test their apps and improve the quality of the tests being developed. The usage of mobile devices such as smartphones and tablets is playing a central role in everyday life. This is also reflected in the growth of the app industry: more than 5 million mobile apps are available in popular marketplaces like Apple App Store and Google Play Store, for a total of over 205 billion downloads and 12 million mobile developers maintaining them [65, 82]. In such a context, developing and delivering high-quality apps represents a strict requirement for developers to stay on the market, keep gaining users, and maintain a high commercial success. All these reasons have led the research community to focus on testing best practices and techniques that could better support mobile developers. As a consequence, mobile developers still mostly conduct testing activities manually. Looking at the available literature, we notice that this aspect has been barely treated so far: while many researchers focused on whether developers use tools for automating testing activities very few of them have analyzed how mobile applications are actually tested. Investigating mobile app testing from the perspective of manually written tests may provide important insights to the research community. Indeed, should mobile apps be well-tested and/or manually written tests be already effective, the urgency of designing automated approaches could be toned down while focusing on how to complement manually written tests and provide developers with information useful to make tests more effective (e.g., which test data should be used to exercise certain boundary conditions).

**METHODOLY:**

We first quantified the number of test classes available for each of the apps in our dataset. Starting from their Github repositories, we cloned the apps locally and, afterwards, we performed an exhaustive search through their packages in order to extract classes having “Test” as prefix or suffix—note that we did not limit the search to classes explicitly using the JUnit framework since some categories of tests (e.g., UI test cases) may be not developed using it. As a result of this search process, we computed the number of test classes and methods per app, which corresponds to the number of test suites and test cases available in a mobile application. Furthermore, we proceeded with a more detailed analysis of these tests that aimed at classifying them according to their granularity (e.g., unit vs. integration) and type (e.g., performance). Given our original dataset, we had to exclude all the apps without tests from this second research question. This process led us to focus on 1,050 mobile apps. To assess the design of the considered test cases we covered two aspects that can characterize their maintainability and understandability. Given our original dataset, we had to exclude all the apps without tests from this second research question. This process led us to focus on 1,050 mobile apps. To assess the design of the considered test cases we covered two aspects that can characterize their maintainability and understandability.

**FINDINGS:**

The first, worrisome result of our study clearly indicated the lack of testing of mobile applications: not only the median number of tests is 2, but also the percentage of apps that do not contain any test is dangerously high (41%). There are multiple factors possibly contributing to this finding. First, our dataset is composed of open-source mobile applications that can be developed under different conditions with respect to other applications: as an example, they can be developed by inexperienced or novice programmers with little knowledge on testing practices. At the same time, the need for testing may be seen as secondary with respect to the development of production code, This is not only true in general, but it seems to affect mobile developers as well. For instance, let consider the case of Acastus Photon,5 an online address/POI search for navigation apps. Looking deeper at its issue tracker and the developer’s comments, we noticed that the developers of the app have consciously postponed some testing activities with the aim of entering the market faster or because of the lack of time to dedicate to testing. According to our findings, most of the test suites present in mobile apps pertain to unit testing, while only a limited amount of them refer to integration and system testing On the one hand, this result somehow confirms the difficulties that developers have when writing these types of test cases. The lack of automated tools and/or support mechanisms likely influences this aspect: as such, our findings represent a call for novel approaches able to work at a higher-level with respect to existing ones that support developers while developing unit tests. Our findings report that most of the tests analyzed are affected by some form of test smells. Previous researches have shown how these problems can turn into critical threats to the effectiveness of tests. The last discussion point relates to the low effectiveness of the test cases analyzed, under all perspectives treated, i.e., code coverage and assertion density. Our results clearly point out that, not only apps are poorly tested, but the available tests are also not effective and likely to miss faults in production code. On the one hand, our findings may be interesting for practitioners and raise their awareness of the status of mobile app testing. On the other hand, this is a further signal of the need for automated solutions that can support developers when performing testing activities.

**RESULT:**

In this paper, we investigated the characteristics of test suites written by developers of mobile applications under three perspectives, namely (1) whether and to what extent these apps are tested and which kind of tests are developed, (2) what is the design quality of the test suites, in terms of code metrics and test smells, and (3) what is the effectiveness of tests, considering assertion density and code coverage. The main results of the study highlight that 59% of the considered apps have at least one test suite; developers mostly test source code to exercise its functionalities, while other types of testing are less widespread. Test smells represent a key problem for most of the test suites, since some of them exhibit characteristics making them possibly flaky. Finally, their effectiveness is low when considering all the computed metrics. Our findings provide a number of implications for researchers, in particular on the need for specific testing tools (e.g., energy or performance approaches) and on the need for studies bringing evidence to developers with respect to the usefulness of testing for the success of mobile applications. A large-scale empirical study on the prominence, design quality, and effectiveness of test cases manually written by developers in the context of mobile applications. A research roadmap on the topic, which can be exploited by fellow researchers to delineate the future steps that may be performed to ease the testing activities of mobile developers. An online appendix [8] containing data and scripts used to conduct our study, and which can be used to further understand our findings and build upon our work.