Exploring the Creation and Added Value of Manufacturing Control Systems for Software Factories

Herwig Mannaert

Normalized Systems Institute
University of Antwerp, Belgium
Email: herwig.mannaert@uantwerp.be

Koen De Cock and Jeroen Faes

Research and Development
NSX bv, Belgium
Email: koen.de.cock@nsx.normalizedsystems.org

***Abstract*—Software engineers have been attempting for many decades to produce or assemble software in a more industrial way. Such an approach is currently often associated with concepts like *Software Product Lines* and *Software Factories*. The monitoring, management, and control of such factories is mainly based on a methodology called *DevOps*. Though current DevOps environments are quite advanced and highly automated, they are based on many different technologies and tools. In this contribution, it is argued that more integrated software manufacturing control systems are needed, similar to control systems in traditional manufacturing. This paper presents a scope, overall architecture and prototype implementation of such an integrated software manufacturing control system. Moreover, several detailed scenarios are elaborated that can leverage such integrated control systems to optimize the operations, and improve both the quality and output of modern software factories.**

***Index Terms*—Software Factories; Software Product Lines; DevOps; Control Systems; Evolvability.**

I. Introduction

The expression *"Software is eating the world"* was formulated in 2011 by Marc Andreessen [1] to convey the trend that many industries were being disrupted and transformed by software. And indeed, more and more major businesses and industries are being run on software systems and delivered as online services. These software systems include *Enterprise Resource Planning (ERP)* systems to design and manage the business processes, *Supervisory Control and Data Acquisition (SCADA)* systems to manage and control production processes in real-time, and *Manufacturing Execution Systems (MES)* to track and document the transformation of raw materials to finished goods, enabling decision-makers to optimize conditions and improve production output. As software systems become more pervasive to manage and control the end-to-end production processes in factories, it seems logical to have or create such control systems for the software systems themselves, i.e., systems to manage and control the building and assembly of software systems in so-called software factories. In this contribution, we explore the creation of such systems to manage and control software manufacturing and assembly.

The remainder of this paper is structured as follows. In Section II, we briefly discuss software factories, the *DevOps* methodology, and situate our approach. In Section III, we describe the scope, overall architecture, and the implementation characteristics of the proposed manufacturing control system for software factories. We present various use cases and types of added value for such an integrated control system in Section IV. Finally, we present some conclusions in Section V.

II. Software Factories and DevOps

A. On Software Factories and Reusability

The idea to produce and/or assemble software in a more industrial way, similar to automated assembly lines in manufacturing, has been pursued for many decades. Such an approach is currently often associated with concepts like *Software Product Lines (SPLs)* and *Software Factories*, but can easily be traced back as far as 1968 to the paper on *mass produced software components* from Doug McIlroy [2]. The concept of *Software Product Lines* has been extensively described by the Carnegie Mellon *Software Engineering Institute (SEI)* [3], and refers in general to software engineering methods, tools and techniques for creating a collection of similar software systems from a shared set of software assets using a common means of production. The characteristic that distinguishes software product lines from previous efforts is predictive versus opportunistic software reuse, as it stresses that software artifacts should only be created when reuse is predicted in one or more products in a well-defined product line [4]. The term *Software Factory* emphasizes the techniques and benefits of traditional manufacturing, and is for instance defined by Greenfield et al. as a software product line that configures extensive tools, processes, and content using a template based on a schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework-based components [5].

The reuse of software artifacts seems crucial in contemporary efforts to realize the benefits of traditional manufacturing through software factories. Nevertheless, the systematic reuse of software artifacts is not a trivial task. Saeed recently argued that software re-usability is not just facing legal issues, but methodological issues as well. Even when only reusing software to save time, and leverage off the specialization of other authors, the end-user must also have the technical

 ISBN: 978-1-68558-098-8

expertise to search, adapt and merge these reusable assets into the larger software infrastructure [6]. We have argued in our previous work that software reuse is even more challenging, and impeded by some fundamental issues related to software evolvability [7] [8]. The sustained technological evolution leads to a continuous sequence of new versions and variants of the software artifacts that need to be reused. These new artifact versions often require changes in their usage that ripple through the entire software structure, causing an impact that is dependent on the size of the system, and limiting the evolvability of software systems [9] [7].

B. From DevOps to Integrated Control Systems

The aim of this contribution is to explore the creation of systems to manage and control the building and assembly of software systems in software factories, similar to SCADA or MES systems in traditional manufacturing. The main approach today in the software development and IT industry to control the building and assembly of software is a methodology called *DevOps*. Used as a set of practices and tools, *DevOps* integrates and automates the work of software development *(Dev)* and IT operations *(Ops)* as a means for improving and shortening the systems development life cycle [10]. It also supports consistency, reliability, and efficiency within the organization, and is usually enabled by a shared code repository or version control. As DevOps researcher Ravi Teja Yarlagadda hypothesizes, *Through DevOps, there is an assumption that all functions can be carried out, controlled, and managed in a central place using a simple code* [11].

Figure 1 presents a traditional overview diagram of a typical DevOps infrastructure environment. While the continuous integration of of the software development and IT operations is represented by the infinity symbol, the representation also contains a typical set of tools and technologies being used in such an infrastructure. We distinguish for example tools for tracking features and user stories (Jira), source control management (Git and Bitbucket), software quality control (SonarQube), automation of build pipelines (Jenkins), automated testing (Cucumber, JUnit), deployment infrastructure (Kubernetes), analytics visualization (Grafana), logging (Graylog), automated deployment (Docker, Ansible), and connecting cloud providers (AWS, Digital Ocean). While the tools in such a *DevOps* or *Continuous Integration Continuous Deployment (CICD)* infrastructure are in general numerous and versatile, there is a clear need for integrated control systems, similar to SCADA or MES systems, encompassing these processes and tools. However, software factories differ significantly from traditional industrial factories, as software is less tangible and the desired control systems need to interface with — often complex — software tools instead of physical equipment.

C. Related Work and Methodology

While academic research is available on various aspects of DevOps, like maturity assessment [12], and management challenges and practices [13], the development of integrated control systems does not seem to be one of them. DevOps platforms are considered to be based on a mix of open source and proprietary software, glued together and built into the platform by a platform team. At the same time, trade publications describe the necessity to breakdown the DevOps phases and tools to increase security and reduce technical debt [14], and acknowledge the need for solutions to scale up DevOps, as nearly a third of DevOps teams' time is spent on manual approaches that are not scalable [15].

The methodology of this paper is based on *Design Science Research* [16], where we design the integrated control system for software factories as an artifact, use a case study to evaluate it in depth in a business environment, and refine the artifact gradually as part of the design search process.

III. A Software Manufacturing Control System

In this section, we elaborate the purpose, scope, architecture, and implementation features of the software manufacturing control system, i.e., the artifact designed in this case study.

A. Purpose and Scope

To design and evaluate the integrated control system artifact, we use the case of the *NSX bv* software factory. It encompasses both the metaprogramming environment and tools to generate applications based on *Normalized Systems Theory (NST)* [7] [8], and actual *Normalized Systems (NS)* applications, i.e., multi-tier web information systems generated in that environment. The various DevOps tools and technologies of the factory correspond to a large extent to those in Figure 1. Though a rather small company, the NSX DevOps environment supports the development and operations of a wide range of heterogeneous and interlinked software artifacts.

- *Run-time libraries* providing basic software utilities to various applications and tools.
- *Expansion resources* consisting of bundles of Normalized Systems code generation modules [8].
- *Web Information Systems*, software applications based on the *Java Enterprise Edition (JEE)* standard.
- *Domain software components*, JEE components that are shared across multiple JEE applications.
- *Integrated Development Tool*, called *μRadiant*, to enable the model-driven development of NS applications.
- *Small tools and plugins* providing additional features in tools like the *μRadiant* or *IntelliJ*.

The various build pipelines, defined in the corresponding software repositories, typically contain the following steps.

- *Expanding* applications or components based on the NST metaprogramming environment.
- *Building* usable libraries, archives, or executables for components, applications, and tools.
- *Unit testing* of various software coding artifacts within the software repositories.
- *Reporting* on the repositories, such as test coverage or software quality metrics.

 ISBN: 978-1-68558-098-8

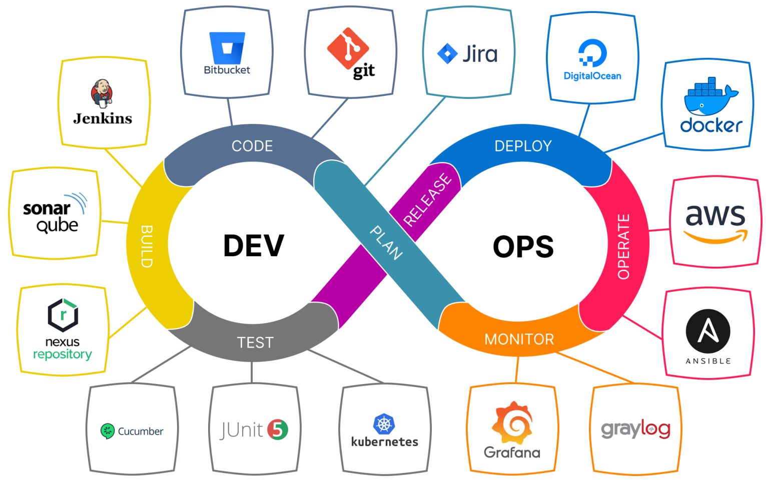

Figure 1. A traditional representation of a typical DevOps infrastructure.

- *Deploying* live instances of applications or tools.
- *Integration testing* invoking live deployments.

Consistent with the overall goal of software factories and product lines, many applications and tools in this DevOps environment are expanded and built by reusing and assembling various other software artifacts that are built in other repositories and pipelines. In order to have an idea of, for instance test coverage and code quality, in a certain version of a software application, we need an overview of these parameters across the versions of all the libraries, expansion resources and components that are being used in that application. Moreover, we need to be able to track the various deployment parameters of all the instances of that version of that application.

This type of functionality, i.e., to manage and control end-to-end the building and assembly of software systems in software factories, is indeed similar to MES systems, i.e., to track and document the transformation of raw materials to finished goods, and SCADA systems, i.e., to manage and control production processes in real-time, in manufacturing. And though almost all the required information is available somewhere in one of the DevOps tools, the integrated overviews and aggregations are not easily accessible.

B. Overall Architecture

The integrated software manufacturing system artifact or prototype for the NSX software factory is implemented itself as a *Normalized Systems (NS)* application, allowing us to take advantage of the *NST* metaprogramming environment. Moreover, as NST was proposed to provide a theoretic foundation to build information systems that provide higher levels of evolvability [9] [7], this should enable us to cope better with the rapidly changing DevOps tools and technologies. NS applications provide the main functionality of information systems through the instantiation of five detailed design patterns or so-called *element structures* [17] [7]:

- Data elements to represent a data or domain entities.
- Action elements to implement computing actions or tasks.
- Workflow elements to orchestrate flows or state machines.
- Connector elements to provide user or service interfaces.
- Trigger elements to to trigger or activate tasks or flows.

At the core of every NS information system is its data model consisting of the various domain entities. A central part of the data model of our software manufacturing control system is represented in Figure 2. As in every software factory, software artifacts are located in versioned *Repositories*. For every repository, automated *Pipelines* can be defined with different steps or *PipelineTargets*, making use of various *BuildTechnologies*. These pipelines produce various types of versioned *Resources*, like libraries, archives, and executables. We distinguish *ApplicationRepositories* corresponding to JEE *Applications* that belong to a certain *Domain*, and *ToolRepositories* for various types of *LeverTools* like plugins, command line tools, or the NS development environment.

The action or task elements serve to import, collect, and or compute various types of data for the software factory control system. Indeed, the manual entering of data in such a system would not only be extremely time consuming, it would also lead to consistency problems. More specifically, types of data that has to be collected or computed, include:

- Versions of applications and lever tools with the corresponding versions of the dependencies or building blocks.
- Aggregated information measures on source repositories, like the number of model entities, or the number and size of source code artifacts.
- Overviews of automated tasks that have been performed in build pipelines with their result status.
- Various quality measures that have been computed for the various applications and tools.
- Aggregated values for the use of different technologies, libraries and expander bundles.

C. Implementation Features

A system or artifact for the monitoring and control of software manufacturing processes should be able to track the

 ISBN: 978-1-68558-098-8

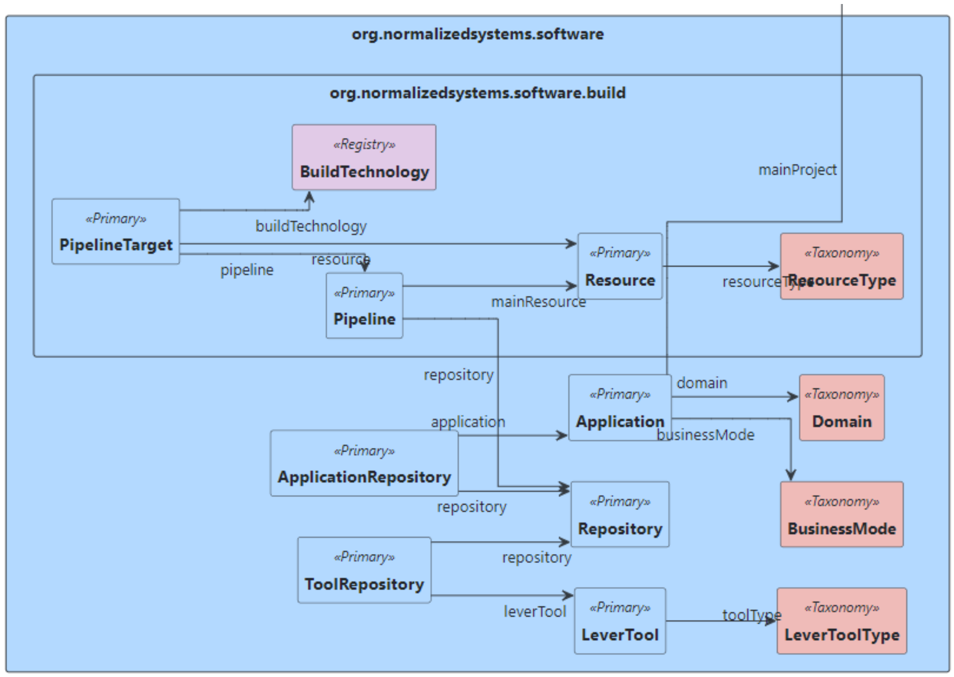

Figure 2. A representation of the data model of the integrated software manufacturng control system.

various parameters and data sets over time. This means that we need to track data over time for most data entities, like sizes of models and custom code, success rates of build processes, or software quality parameters. Therefore, so-called *history or log tables* are a crucial part of the data model. In the NS metaprogramming environment, *expanders* or code generators exist to automatically add —and even populate— an history element for every data element. These history tables can then be represented in graphs and analyzed over time, looking for possible improvements in productivity and/or output quality.

A large part of the relevant data for the software manufacturing control system is already present or computed in one of the many tools or technologies represented in Figure 1. This implies that the automated collection and or computation of software factory data in automated tasks integrates with these tools and technologies, such as *Bitbucket* repositories, *Maven* dependency declarations, *Jenkins* build engines, and *SonarQube* quality analyzers. In accordance with NST, there is a decoupling between the functionality of the data collection in the the task element, e.g., build engine results or quality measurements, and the actual implementation (class) of the task element, e.g., getting data from *Jenkins* or *SonarQube*. In this way, the software manufacturing control system is able to support additional versions or variants of these tools and technologies with limited impact.

IV. Toward a Control Layer for Software Factory Improvements

As stated in Section I, by tracking and documenting the transformation of raw materials to finished goods, *MES* enable decision-makers to optimize conditions and improve production output. In the same way, a software manufacturing control system should provide an analysis platform and control layer to improve and optimize various aspects and characteristics of the software factory operations and output. In this section, we discuss some use cases and their added value, as they are being developed as part of the iterative case-based design process.

A. Monitoring Evolutions over Time

A first avenue to optimize and improve the output and quality of the software factory, is to monitor the evolution of certain parameters over time. As explained in [8], NS information systems distinguish between software skeletons, instantiations of element structures generated by modular code generators or so-called expanders, and custom code being additional software artifacts or classes, i.e., *extensions*, or code snippets added to the generated artifacts or classes, i.e., *insertions*. From a quality and evolvability point of view, it is important to monitor the amount, size, and location of these extensions and insertions. As an example, some sample graphs are shown in Figure 3. They represent, for a specific information system, the evolution of the total amount of insertion snippets, and the total size of those insertion snippets. As done for the second graph, these values can be made relative with respect to the evolving size of the model, i.e., the number of element structures.

This type of monitoring, based on automated data collection from the *Bitbucket* repositories, has been performed for quite some time in the NSX software factory. It has provided valuable insights into the actual project phases when such custom code typically grew fast, and the software layers

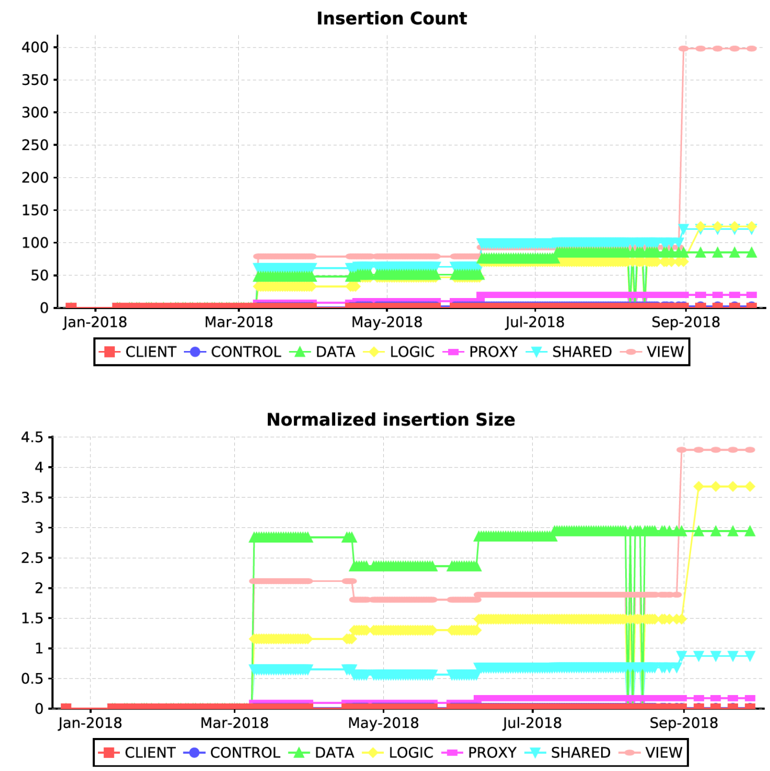

Figure 3. Sample graphs monitoring the amount of custom insertions and the normalized total size of insertions.

where such custom snippets where needed the most. This last parameter provided an indication in which layer the development of the code generators should be prioritized to provide more out of the box functionality. Of course, other types of software systems and factories, without the typical NS distinction between element structures and custom code, should monitor other structural measures to improve software structure and productivity.

While this has been integrated in the software manufacturing control system, the monitoring over time is not limited to the source code repositories. History tables are also being created for success rates of build pipelines, quality measures of the custom source code, test overage percentages and numbers of failed tests, numbers of live system deployments, etc. Of course, this implies the integration with various DevOps tools and technologies.

B. Aggregating over Manufacturing Chains

It is often considered to be a crucial characteristic of software factories and software product lines that software artifacts should only be created when their reuse is predicted in one or more products [4]. And for instance in the NSX software factory, a typical JEE application uses various other software artifacts produced by the factory, such as:

- Several *runtime libraries* providing various utilities like file handling or protocol adapters.
- Several *reusable components* supporting more generic functionality like workflows or notifications, and/or providing more domain-specific building blocks such as project planning or human resource benefits.
- Several *expander bundles* that are used during the expansion of the application, such as the expanders to generate the instances of the NST element structures, or extensions such as the *Relational State Transfer (REST)* interfaces.

These artifacts are in general stored in other repositories and built in other CICD pipelines. And while the dependency on the code generation modules may be specific to NS applications, the dependencies on various runtime libraries and domain components is valid for nearly every software factory.

While parameters related to, for instance test coverage and code quality, can be monitored for every individual software artifact that is created in the factory, it seems quite relevant to offer instant overviews and aggregations of these parameters for all artifacts that are part of a specific aggregated artifact, such as a JEE application. While obviously being relevant to the customers using or licensing such an application, this also enables the optimization and improvement of the overall quality of the factory itself. Indeed, it allows to quickly identify the weak links in such aggregated artifacts, and to prioritize these software artifacts for improvements.

C. Tracking Technology Use Across Projects

Software applications are in general dependent on multiple external artifacts and technologies, e.g., libraries and plugins, that are build outside the software factory by commercial software vendors or in open source projects. While these

 ISBN: 978-1-68558-098-8

dependencies are available in configuration files, it is important to surface overviews and aggregations of these dependencies. Such overviews and their added value include:

- immediate overviews of the impacted applications when a vulnerability is detected in a library or technology.
- straightforward assessments of the impact when retiring a certain (version of a) technology.
- regular evaluations of the usage and adoption rate of libraries or expander bundles from the factory itself.

Obviously, such integrated information would also support decisions concerning internal resource allocation, both for supporting both internal and external technologies.

V. Discussion and Conclusion

For many years, software engineers have strived to produce and/or assemble software in a more industrial way. In today's software factories, building and assembling software systems is mainly controlled using a methodology called *DevOps*. These *DevOps* environments are quite advanced and highly automated, but are in general based on many different technologies and tools. As previously experienced in the automation of business processes and traditional manufacturing, this often leads to a need for more integrated systems. In this contribution, we have explored the creation of an integrated software manufacturing control system, similar to SCADA or MES systems in traditional manufacturing.

As part of a case-based design science approach, we have presented a functional scope and overall architecture for such a software manufacturing control system, and have described the design and prototype implementation of the artifact for the software factory case. This software manufacturing control system prototype does not provide fundamentally new information, but collects, aggregates and integrates information over time, across various repositories and build pipelines, and from different DevOps tools and technologies. Therefore, this control system does not provide new possibilities per se to optimize processes and improve output in software factories, as this can be done today by analyzing in detail the data produced by the various tools. However, aggregating and providing this information in nearly real-time, offers the opportunity to fundamentally reduce the lag times for such optimizations and improvements. Though the design as a search process is still ongoing, we have presented some use cases where the added value was validated in the case study.

Exploring the creation of such a software manufacturing control system is believed to make some contributions. First, we have identified and validated a need for integrated control in today's state of the art automated DevOps environments. Second, we have designed an architecture that enables the rather straightforward creation of such integrated software manufacturing control systems in most contemporary software factories. Third, we have described and validated a number of detailed scenarios that can leverage such an integrated control system to improve the output of such software factories.

Next to these contributions, it is clear that this explorative paper is also subject to a number of limitations. First, the case-based approach means that the integrated system has been created for a single software factory, though this factory does include for instance code generators. Second, the major part of the added value through optimizations and improvements, enabled by the drastic reduction of the lag times in the control processes, has yet to be confirmed empirically. However, its design has been validated by actors in our case study, and we are planning the empirical validation in the near future.

References

[1] A. Marc, "Why Software Is Eating the World," URL: https://a16z.com/2011/08/20/why-software-is-eating-the-world/, 2011, [accessed: 2023-07-27].

[2] M. D. McIlroy, "Mass produced software components," in Proceedings of NATO Software Engineering Conference, Garmisch, Germany, October 1968, pp. 138–155.

[3] S. E. Institute, "Software Product Lines Collection," URL: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513819, 2023, [accessed: 2023-07-27].

[4] C. Krueger, "Introduction to the emerging practice software product line development," Methods and Tools, vol. 14, no. 3, 2006, pp. 3–15.

[5] J. Greenfield, K. Short, and S. Cook, Steve; Kent, Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, 2004.

[6] T. Saeed, "Current issues in software re-usability: A critical review of the methodological & legal issues," Journal of Software Engineering and Applications, vol. 13, no. 9, 2020, pp. 206–217.

[7] H. Mannaert, J. Verelst, and P. De Bruyn, Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design. Koppa, 2016.

[8] H. Mannaert, K. De Cock, P. Uhnak, and J. Verelst, "On the realization of meta-circular code generation and two-sided collaborative metaprogramming," International Journal on Advances in Software, no. 13, 2020, pp. 149–159.

[9] H. Mannaert, J. Verelst, and K. Ven, "The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability," Science of Computer Programming, vol. 76, no. 12, 2011, pp. 1210–1222, special Issue on Software Evolution, Adaptability and Variability.

[10] M. Courtemanche, E. Mell, and A. S. Gills, "What Is DevOps? The Ultimate Guide," URL: https://www.techtarget.com/searchitoperations/definition/DevOps, 2023, [accessed: 2023-07-27].

[11] R. T. Yarlagadda, "Devops and its practices," International Journal of Creative Research Thoughts (IJCRT), vol. 9, no. 3, 2021, pp. 111–119.

[12] A. Kumar, M. Nadeem, and S. M., "Assessing the maturity of devops practices in software industry: An empirical study of helena2 dataset," in Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering (EASE 22), 2022, pp. 428—-432.

[13] S. M. Faaiz, S. U. R. Khan, S. Hussain, W. Wang, and N. Ibrahim, "A study on management challenges and practices in devops," in Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE 23), 2023, pp. 430—-437.

[14] M. B. et al., "How to Select DevSecOps Tools for Secure Software Delivery," URL: https://www.gartner.com/en/documents/4131199, 2023, [accessed: 2023-10-10].

[15] Dynatrace, "Observability and security convergence: Enabling faster, more secure innovation in the cloud," URL: https://assets.dynatrace.com/en/docs/report/bae1393-rp-2023-global-cio-report-observability-security-convergence.pdf, 2023, [accessed: 2023-10-10].

[16] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," MIS Quarterly, vol. 28, no. 1, 2004, pp. 75–105.

[17] H. Mannaert, J. Verelst, and K. Ven, "Towards evolvable software architectures based on systems theoretic stability," Software: Practice and Experience, vol. 42, no. 1, 2012, pp. 89–116.