DEVOPS AND MODERN SOFTWARE DELIVERY

Damir Solajic*,
Anamarija Petrović

Levi9 Technology Services,
Novi Sad, Serbia

Abstract:
Delivering software is often a long, difficult and risky process. Defects and integration issues pop-up at the very last moment and cause dissatisfaction to end users, the development teams and business stakeholders. Furthermore, lack of collaboration between different teams generally results in the implementation of wrong functionality, integration and deployment errors and finger-pointing. The goal of this paper is to summarize the best DevOps practices relevant to software delivery. These practices are a result of a systematic and continuous improvement of the software delivery process and range of behavioral and cultural changes in the way of working and technical improvements in the software delivery process.

Keywords:
DevOps, Lean, Continuous Delivery.

1. INTRODUCTION

In the traditional organization of the software development process, a stream of work flows from an idea from business and marketing departments to product management and design departments who translate this idea into business requirements and specifications. These product requirements are then handed over to software development teams who are converting them into software code. The running code is inspected by a quality assurance, who test it against their interpretation of the original product requirements. The IT Operations team performs preparation of appropriate environments and deployment of the code. At the very end, often too late, work gets tested on security aspects by an Infosec team.

Furthermore, Development and IT Operations have quite opposed goals. Development has a prime objective to introduce new features and therefore bring change into the system, while the goal of IT Operations is to preserve stability and prevent change as a possible cause of instability. All these teams are usually in the separate organizational departments, behaving like silo's, trying to protect their realms. This conflict leads to poor software and service quality and bad customer experience. Such a way of working very much lacks attention to what should be the common goal, which is the fast and continuous delivery of valuable software to customers [1].

Correspondence:
Damir Solajic

e-mail:
d.solajic@levi9.com

DOI: 10.15308/Sinteza-2019-360-368

This process has at least five critical handover moments, where the information is being transitioned between different departments and distorted due to different interpretations. From a time perspective, this traditional silo organization causes significantly higher lead times from a business idea to implementation, due to a proportional delay in every silo. If we define Wait Time as the ratio of Busy vs. Idle, then at 50% utilization wait time will be 1 unit of time. At 90% this will be nine units. With earlier mentioned five handover moments total wait time would be 45 units of time [2].

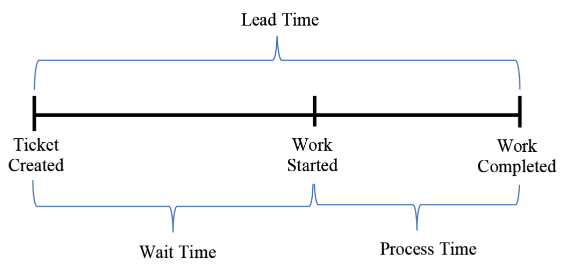

Fig. 1. Definition of Lead Time as a sum of Wait Time and Process Time

"When people are trapped in this downward spiral for years, especially those who are downstream of Development, they often feel stuck in a system that preordains failure and leaves them powerless to change the outcomes. This powerlessness is often followed by burnout, with the associated feelings of fatigue, cynicism, and even hopelessness and despair." [2]

We must change the way of working, and we believe that DevOps practices are showing us the best way forward.

High performing IT organizations that are applying DevOps practices are outperforming their less performing competition by deploying code thirty times more frequent with sixty times higher success rate, having two hundred times faster lead time for implementation of changes and 168 times faster mean time to restore the service [3]. These companies have higher growth rates, but also higher employee satisfaction and lower rates of employee burnout. They manage to consistently provide stable, reliable and secure service to their customers.

DevOps approach enables organizations to create a stable way of working, where small poly-skilled teams can efficiently and independently develop, test and deploy code and add value to customers, quickly, safely, securely, and reliably. DevOps approach allows organizations to maximize productivity, enable organizational learning, create high-trust, high employee satisfaction collaborative culture that helps them win in the competitive market.

The DevOps practices are complementing Lean and Agile Software Development practices and can be classified into three areas [4]:

- Ensuring smooth and fast Flow of work from product design and development to operations and ultimately to the customer
- Enabling fast Feedback from operations to development, to facilitate quick detection, recovery and, in the end, to prevent problems from happening again
- Creating a high-trust culture of Continuous Improvement that allows initiative and experimentation and embeds a culture of organizational learning.

"DevOps defines technology value stream as the process required to convert a business idea into a technology-enabled service that delivers value to the customer" [2], so the primary task of Flow, Feedback and Continuous Improvement is to ensure fast and reliable delivery of useful software to the customer.

2. FLOW

The goal of DevOps practices of Flow is to ensure smooth flow of work from Product Design, through Development into the Operations. Since the software code has value only when it is in the production and used by the customers, the goal is to increase flow and reduce lead time from a business idea to software deployed in the production.

First and foremost, to be able to organize the work, we need to make all work in the value stream visible: product backlog items, software defects, production incidents, service requests, everything. We can organize issues in the form of Kanban or Scrum boards and promote a pull system of work. Furthermore, this approach enables us to perform value stream analysis which gives us a starting point for our improvement actions. One way of making this analysis is a value stream map presented on a figure below.

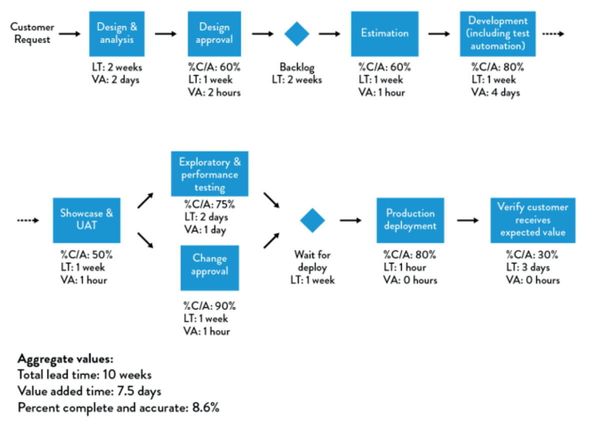

Fig. 2. Value Stream Map [4]

Just like in the manufacturing, based on the Theory of Constraints, to improve the flow of the process, we should subordinate everything to identifying and improving the bottleneck. Improving flow at any work station before the bottleneck would cause a further pile-up of the inventory at the bottleneck (e.g., un-checked-in code) while improving anything behind it, will cause further starvation of the work stations behind (e.g., nothing to test). Such an improvement process is in principle a continuous improvement cycle, similar to the well-known Plan-Do-Check-Act Deming cycle.

The causes that disrupt productive flow in the technology value stream, usually called sources of waste, are: defects, partially done work, waiting, motion, task switching, extra processes that do not add value, extra features, non-standard or manual work and heroics [4]. The goal of a continuous improvement cycle is to make these wastes and hardships visible and to work on eliminating them.

We used the following techniques in our continuous endeavor to reduce waste and improve flow.

Reduce the Number of Handoffs

Initially, we integrated software development and quality assurance teams into one team. Not only personnel were integrated, but also activities. Testing activities are executed from the very start of the delivery iterations with for instance preparation of test cases. Introduction of automated testing not only significantly reduced wait time but also contributed to a way of working where the whole team participates in the testing activities, thus enabling a smoother flow and greater flexibility.

The second step encompassed improving collaboration between Development and IT Operations. Depending on the case, we used two approaches: separate functional teams or integrated feature teams (sometimes named as market-oriented teams). With the functional team approach, IT Operations team is responsible for enabling full self-service possibilities for software development teams, so that software development teams can perform all activities without a need to make a request to IT Operations. These activities especially encompass provisioning of environments and deployment activities which were critical handover moments due to a large amount of waste created (due to wait and defects). For all other activities, each software development team has a designated point of contact in IT Operations team. This contact is responsible for understanding the scope of the work of the development team and ensuring that the self-service platform is serving the needs of the development team.

In the feature teams approach, IT Operations is an integral part of the software development teams, and their activities are planned and executed together with all the other activities. Furthermore, the natural exchange of the knowledge contributed to the fact that developers started performing deployments, while system engineers understood better the functioning of the applications.

At the third step, we focused on improving the collaboration with product management. The goal of this step was to reach a common understanding of each other's work and to improve the quality of requirements and specifications. We applied the techniques of Specification by Example [5]: specifying collaboratively (e.g., user story mapping sessions, refinement sessions), illustrating using examples and living documentation that enabled building the right things and building them right.

Reduce Batch Sizes

The Flow can be increased by reducing intervals of work, minimizing batch sizes, building quality in from the very start and preventing defects flowing to downstream work centers. Batch size proved to have a leading role in increasing flow, consequently reducing lead time and improving quality.

In the software development value stream, batch size equals to the amount of an undelivered code. Large batch sizes, due to integration issues, directly increase lead time and decrease quality. Lower quality means massive disruptions to downstream work centers being Quality Assurance, IT Operations, and Security. On the other side, reducing batch sizes leads to shorter lead times, enables faster error detection, while the defects are still small, which, by default, implies shorter recovery times. Modern software delivery practices prefer optimizing Mean Time to Restore (MTTR) more than Mean Time Between Failures (MTBF), which in principle means being able to recover fast from failures that will inevitably happen, compared to of minimizing the number of defects. Reducing batch sizes is achieved by shortening delivery iterations and by integrating code frequently.

Limit Work in Progress

Work in progress is one of the most significant sources of waste in the software development process. Unfinished tasks are an inventory that is never used but creates costs. Without being managed, work in progress is piling up, causes multitasking and task switching which consequently severely degrades a flow. Work in progress needs to be limited to ensure it is getting finished. Limiting work in progress makes it also easier to identify problems which are preventing work completion. If, for example, limiting work in progress causes that we do not have any work, although it might be tempting to take a new task, it would be smarter to see what is causing a delay in upstream work center and fix that problem.

Continuously Identify and Improve Constraints and Eliminate Waste

As stated above, a focus should always be on one and only one constraint. We need to identify that constraint and work on improving it continually. Improving on anything besides that constraint is a pure waste at that moment in time. Usual sources of constraints in the software development process are unclear requirements, disruption in environment provisioning, deployments, test execution, and non-evolving architecture. Furthermore, we should continuously work on removing waste from our process, firstly minimizing rework by preventing the flow of defects to downstream work centers.

In all cases above, it proved, that although organization structure is important, the culture is fundamental. The culture is a way how people act and react, especially in times of need. Ownership of the product and improvement mindset are critical success factors for any team. The teams that took ownership, homogenized, continuously improved, team members helped each other, acted regardless of individual competencies or preferences and at the end successfully delivered high quality.

Continuous Delivery

Technical practices of Flow implemented through Continuous Delivery pipeline [6] enable integrated and automated flow of work from the keyboards of the developers to the production environment. These practices establish a repeatable and reliable process for software development teams to continually check-in code changes in the version control system, perform automated tests against it and deploy it to production.

We can divide these practices into five areas:

- Version Control practices
- Continuous Integration
- Test Automation
- Infrastructure Automation
- Automated Deployment

All source code (application, tests, configuration, data, infrastructure) changes are checked-in to a single Version Control System. All code changes are performed either on the master branch (so-called trunk-based development) or short-lived feature branches. Binaries are kept in the artifact repository, but also can be recreated from the source code. Practices like feature toggles enable integrating the code without affecting other functionalities.

Each check-in (potentially) triggers a Continuous Integration build. Each integration is verified by an automated build and by unit and integration tests execution to detect component and integration errors as quickly as possible. This approach leads to a reduced number of integration problems and allows the teams to develop software more rapidly. In the case of a broken build, all ongoing activities stop until the problem is solved and the build becomes releasable again.

Test Automation as a practice enables automated execution of tests by comparing actual and expected results. Test Automation encompasses unit, integration, functional, performance and security tests and reduces the need for manual testing to a minimum. It requires right skillset from both testers as well as developers and their continuous teamwork, but also requires quality embedded in the architecture of the software itself.

Infrastructure Automation enables automated on-demand provisioning and configuration of hosting environments. With modern "Infrastructure as a Code" tools like Puppet or Chef and public or private cloud and container possibilities, this enables us to quickly spin-up any environment from the code, run our tests and deallocate the environment when automated tests finish. Even more, we can run as many testing environments as we need and ease resource dependencies. Software development teams can provision production like environment very early and continuously ensure that application is running reliably. With this approach, we guarantee that all environment changes are executed from the code, therefore disabling possibility for manual errors. Faulty machines or environments can quickly be decommissioned, and put new ones in their place since it is easier to rebuild than to repair.

Finally, Automated Deployment covers the so-called "last mile" of the deployment pipeline. This part of the process can be fully or semi-automated. Full automation of the deployments is usually used in the case of testing environments, while in the case of production environment we potentially want to keep the additional level of control which includes the manual step of exploratory or smoke testing before an automated deployment. Deployment process needs to be an automated self-service so that any team member can deploy an application to any environment, reliably and without fear of making a mistake. It is crucial that we decouple deployments (installation of a specific version of the software to a particular environment) from releases (exposing new functionality to customers). Executing deployments should be easy, stress-free, repeatable activity. Releasing new features requires a careful selection of the right strategy to make it such. Two groups of release patterns exist, environment-based and application-based. Environment-based release pattern requires that different versions of an application exist in different environments. The actual release is then executed merely by making the desired environment available to all or only a group of users (so-called blue-green releases and canary releases). With application-based release patterns, the new release is codified (with feature toggles) in the application and configuration code and made available with the right code changes (so-called dark launching).

Tooling is not a solution for Continuous Delivery by itself. It is just a mean that should help to create an integrated and automated solution, that in combination with the right culture of taking ownership creates a fast and smooth flow of work from the business idea to the successful product used by end customers. Most importantly, to make Continuous Delivery a success, Development, Test and IT Operations teams need to work together as one delivery team from the very start.

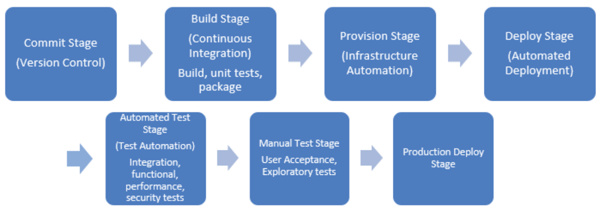

Fig. 3. Example Continuous Delivery Pipeline

Based on our experience, software and system architecture choices very much determine our ability to set continuous delivery in practice. Systems should be architected for testability, deployability, and monitorability and architecture itself should be able to evolve based on the needs of the system. Modular, loosely-coupled, well-encapsulated architectures, with well-defined interfaces exposed through APIs, have low architectural entropy and enable a much higher degree of adaptability and agility in general. Legacy, monolithic systems require much more effort to reach a satisfactory level of automation, but the patterns like strangler application pattern or branching by abstraction enable incrementally evolving a whole system, step-by-step leading to an entirely new one. These patterns allow the evolutionary design of the application architecture while allowing everybody to work on the same source code.

3. FEEDBACK

The goal of DevOps practices of Feedback is to enable fast and constant feedback loops, with the aim to create a more reliable, secure and resilient system. Our continuous care should be to detect problems while they are still small and easy to fix, to find them before they are visible to our customers and to prevent major system outage. In the end, we need to learn from problems and embed these learnings to future work.

We can do this only by creating, amplifying and shortening feedback loops so that we see the problems as they occur. Failures are inevitable in any complex system, so we must design a safe system of work, where work is done without fear, with confidence that errors will be detected quickly, long before serious consequences.

Identifying problems as they occur enables us to validate or invalidate possible causes quickly and to be faster and therefore cheaper in finding and fixing the issues.

Solving Problems as They Occur

In the case when the issues occur, whether these are production issues or broken build during continuous integration, we should team-up to solve the problem quickly. Like in manufacturing, with Andon cord of Toyota Production System, the production line should be stopped until the problem is solved. Upon solution, we should share learnings throughout the organization to further improve the processes. This approach, compared to fixing when we have time, prevents the problem, as a source of waste, going to a downstream work center, where the cost and effort to repair it would be significantly higher. Moreover, it prevents the accumulation of technical debt.

Stopping all other work and especially preventing the start of new work before the problem is solved, ensures that new errors are not introduced into the system until current ones are resolved. The work center could potentially have the same problem in the next operation if we do not address it right away. The issues are often a result of specific circumstances in the complex systems, and, if not treated immediately, it is likely to be impossible to reconstruct particular events that were leading to them and therefore much more difficult to investigate the cause and find the solution. Good version control commenting practices and change logs prove to be an efficient instrument for tracing the source of problems.

Team Ownership

We reinforce the feedback loop if Development teams participate in the support activities together with IT Operations and others. Support activities also include standby support during out-of-office hours. The risk of being woken up in the middle of the night makes everybody being much more diligent. In such cases, not only developers better understand the hardships of downstream work centers, but also more quality gets built-in. At many high-performing organizations with a functional way of working, development teams are responsible for running their services in the production until the services are stable enough to be transitioned to IT Operations. With this approach we push quality closer to the source, encourage teams to automate their processes and above all, to take ownership and responsibility for the product and services they are providing. The team should additionally use techniques of contextual inquiry, where they sit together with the end users to understand how they use the product and what are possible improvements from the usability perspective.

Peer-reviews and Change Coordination

Peer review process proved to be a useful practice of feedback that results with an increase of quality of the code and enables knowledge sharing. Modern version control repositories enable peer-review through pull request process where each check-in gets reviewed by a predefined number of designated experts. This process requires discipline to keep review batch sizes small so that review is easier to plan and perform. It also enables moving away from inspections and approvals which are typical for low-trust environments with command-and-control cultures. High performing organizations rely more on peer reviews than on external approval of changes [3].

In multi-team environments, it is imperative to ensure change coordination and scheduling of changes. When the deployment batch size is more significant, change success rates go down, while a number of incidents and recovery time grow. Therefore, especially in a multi-team environment, it is essential to keep different teams working on the same tact, having them synchronized and managing the dependencies between them well. A possible approach is to have every team represented in the separate integration team that takes care of coordination, integration, and synchronization of work.

Create Centralized Telemetry

Technical practices of Feedback require creating means of centralized monitoring for gathering telemetry. We establish centralized monitoring with monitoring server(s), which collect different data (events, logs, and metrics) using agents running on the monitored objects (virtual machines, containers, physical equipment) or by entering data from different sources through an API. Modern automation techniques enable that each machine or application can take care of parameters they want to be monitored on and make sure that either they are automatically registered or dynamically discovered by the monitoring server.

The gathered data we call telemetry and classify it to:

- Business telemetry (e.g., number of sales transactions, number of new users, number of logins)
- Application telemetry (application health/faults, transaction times, applications logs, external systems)
- Client telemetry (application errors and crashes on the client side)
- Environment telemetry (operating system, networking, database, storage, web traffic, CPU, memory)
- Deployment Pipeline telemetry (deployment frequency, deployment status, static code analysis metrics, code coverage metrics)

Telemetry helps us to pinpoint and solve the problems faster, whether it is a defect in the application code, in the environment, data or external. It enables a disciplined approach to problem-solving, where we can react based on the facts and not on the notion (e.g., restarting a server whenever we have a problem, without finding the underlying cause). Solving problems should also encompass adding new telemetry, that will enable easier discovery and prevention of such issues in the future. This process should be repeated for ever-weaker failure signals (incidents and near misses) with the aim to achieve pro-active problem prevention.

The telemetry should be accessible and shared with the entire value stream. Retrieving telemetry information from the telemetry system should be self-service, through dashboards and APIs and not by opening tickets for IT Operations. Adding business and application telemetry should be easy through code instrumentation and APIs. Furthermore, telemetry information should be put out on display in highly visible locations, through so-called information radiators. Having telemetry accessible and visible enables us to quickly notice the problem, structurally solve it based on the facts and not only have a significantly better time to recover, but also strengthen the relationship between Development and Operations by creating empathy and trust between upstream and downstream work centers.

Continuously Analyze and Improve Telemetry

Different graphical visualization tools can visually represent telemetry data. These tools, as well as different statistical analysis tools, can help us to analyze and use telemetry data for various purposes which range from trend analysis and predictions (e.g., the prediction for auto-scaling), outlier and anomaly detection, to smart, pro-active alerting and escalation. Furthermore, these tools enable us to cross-reference different metrics to find correlations between business outcomes or application defects and application, environment or deployment telemetry. Usage of simple statistical methods for data with normal distribution can prove to be quite useful for pro-active alerting and automated actions in the case of outlier detection (detection of nodes that are different from others). More complex algorithms might be needed to analyze the data with non-normal distribution to achieve good anomaly detection or predictive analysis.

Experimenting

When we have our continuous delivery and telemetry system with relevant data in place, we become much more agile. Agility enables us to perform user research by experimenting in production and quickly validate or invalidate our business ideas. Without user research, there is a high chance (research [4] shows 2 out of 3) that our features deliver no value to our organization, while they make our codebase more complex, difficult to maintain and change. Furthermore, “the effort to build these features is made at the expense of delivering features that would deliver value” [4].

A/B testing or hypothesis-driven development techniques enable us to implement business ideas in minimalistic form, quickly try them in production, compare the outcomes with expectations and make informed decisions. We can do this without risk, in some cases even in a fully automated fashion, since the telemetry system and automated release techniques safeguard us from negative outcomes. A/B or split testing is a well-known marketing technique that enables validation of an idea by comparing control and treatment specimen. When we release a new feature (treatment) to a targeted group of users (canary releasing), we can compare the telemetry data with control specimen and decide on the validity of our business idea before further investment in its development. If the idea proves to be invalid, we can switch it off with feature flags. If the implementation is faulty, we can decide to fix forward or to rollback.

4. CONTINUOUS LEARNING AND IMPROVEMENT

Finally, DevOps practice of Continuous Learning and Improvement creates the culture of continuous learning and experimentation, constant creation of individual knowledge which is turned into organizational knowledge with the primary goal to improve our service. Only high-trust culture, without room for fear and punishment in the cases of errors and mistakes, can be a constructive ground for continuous improvement. This culture leads to the creation of safe systems of work, that result in high-quality software products with embedded resilience and safety.

Enable Organizational Learning

Safe systems of work mean that when incidents occur, we look for long-term structural solutions that prevent such incidents from happening again. We do this through a disciplined, systematic approach by avoiding any blaming of the ones potentially responsible for the problem. We should also avoid creating more processes and procedures that would "prevent" such problems from happening. Blaming and bureaucracy do not prevent nor solve the problems, but they do cause fear, which results in issues usually remaining hidden until real disasters happen. We must avoid it and instead we should define failure as an opportunity for learning and improvement.

After the solution of the incident, we conduct blameless post-mortem sessions, where we discuss what led to the incident and what measures, technical and behavioral, we can take to prevent it from happening again. We use this approach for serious to less serious issues, eventually getting to near-misses and being able to prevent problems before they occur. We record these sessions into our knowledge database, promote this knowledge into the organization and embed it as broader organizational learning. Knowledge database should be easily searchable and accessible for everyone. Furthermore, team channels or chat rooms proved to be useful platforms for sharing knowledge for resolution of incidents and broader knowledge exchange, so logs from those sessions should also be accessible. All these local discoveries should be converted into global knowledge and global improvements.

In the end, "code is the ultimate truth" so we should keep knowledge in the version control repositories accessible, not only the software code, tests, and configuration but also living documentation [5], in the form of codified specifications expressed in Gherkin test cases and automated tests.

Institutionalize the Improvement of Daily Work

"In the absence of improvements, processes do not stay the same – due to chaos and entropy processes degrade over time." [2]

When we avoid fixing the problems structurally, continuously patching them with workarounds, our problems and technical debt accumulate. Eventually, technical debt makes any new work quite expensive or, even worse, prevent the organization from completing any further work. Reserving time to deal with technical debt, adding new automated tests to detect boundary conditions, adding new production telemetry, identifying categories of changes that require peer-review, organizing Kaizen Blitzes or Hackathons and conducting exercises with Game-Days proved to be useful instruments to continuously improve the application and level of our service.

Based on our experience some 5 – 20% of the software development team time should continuously be used to pay the technical debt. This time should be wisely used to introduce improvements and innovation. We should always use a chance for "opportunistic refactoring" and "always leave the code a little bit better than we found it" [7]. While Kaizen Blitz exercises enable teams to self-organize and work for a predefined amount of time on fixing any problem they find essential, Game-Day testing allows them to improve resilience, reliability, and stability of their applications by introducing instability and failure into the system. On the other side, Hackathons enable teams to work on their innovative ideas.

In the end, whatever we do, we should value the improvement of our daily work more than a daily work itself. It is a responsibility of leaders to create conditions in which their teams can thrive. They should promote the value of continuous learning, disciplined problem-solving, calculated risk-taking, continuous questioning and experimentation over just being careful.

5. CONCLUSION

Based on our extensive experience, we are fully convinced that the DevOps approach to software delivery is a modern and comprehensive approach that results in high-quality software products and customer satisfaction. These practices require a change in the technology aspect of the value stream, through modernization, automation, and innovation. Even more importantly, it needs a change in the cultural and behavioral approach to software development as a practice, through taking ownership of the work, responsibility for our products and services, optimizing work streams and teaming-up during problems. When applied together, these practices lead to the state-of-the-art software development process, quality products, happy employees and satisfied customers.

REFERENCES

[1] Agile Manifesto, https://agilemanifesto.org/principles.html

[2] G. Kim, K. Behr, G. Spafford, The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win, IT Revolution Press; 1st edition, 2013

[3] The State of DevOps Reports 2014 - 2018, https://puppet.com/resources/whitepaper/state-of-devops-report

[4] G. Kim, P. Debois, J. Willis, J. Humble, J. Allspaw, The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations, IT Revolution Press, 2016

[5] G. Adzic, Specification by Example: How Successful Teams Deliver the Right Software, Manning Publications; 1st edition, 2011

[6] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley Professional; 1st edition, 2010

[7] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall; 1st edition, 2008