

21.08.2017



eficode

VOLVO IT: SCQ

DEVOPS DEVELOPMENT REPORT

INDEX

- 1. INTRODUCTION**

- 2. SUMMARY**

- 3. KEY OBSERVATIONS**

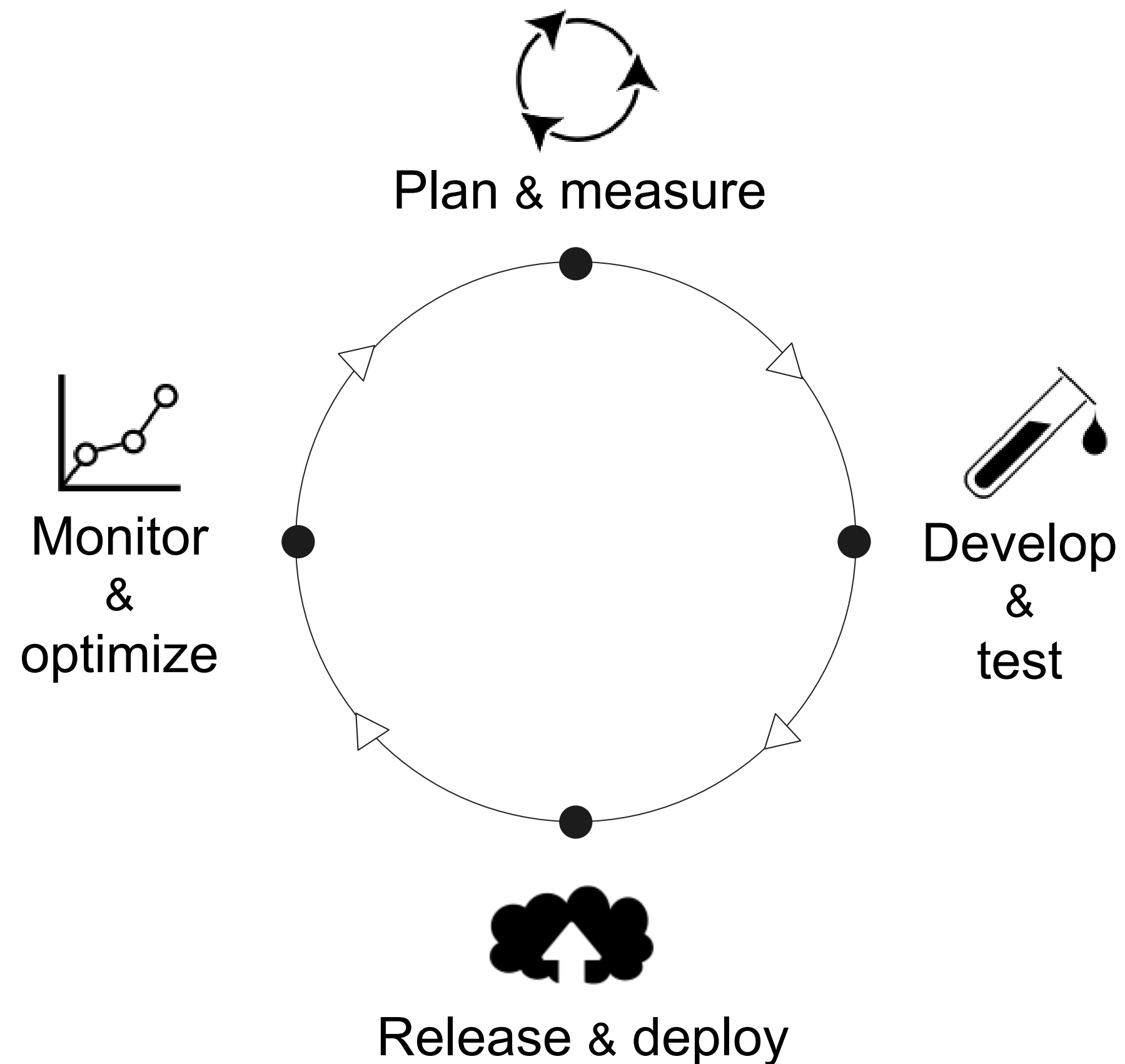
- 4. IMPROVEMENT SUGGESTIONS,
BENEFITS & ROADMAP**

- 5. APPENDIXES**

INTRODUCTION

DEVOPS IN A NUTSHELL

Devops enables continuous software delivery

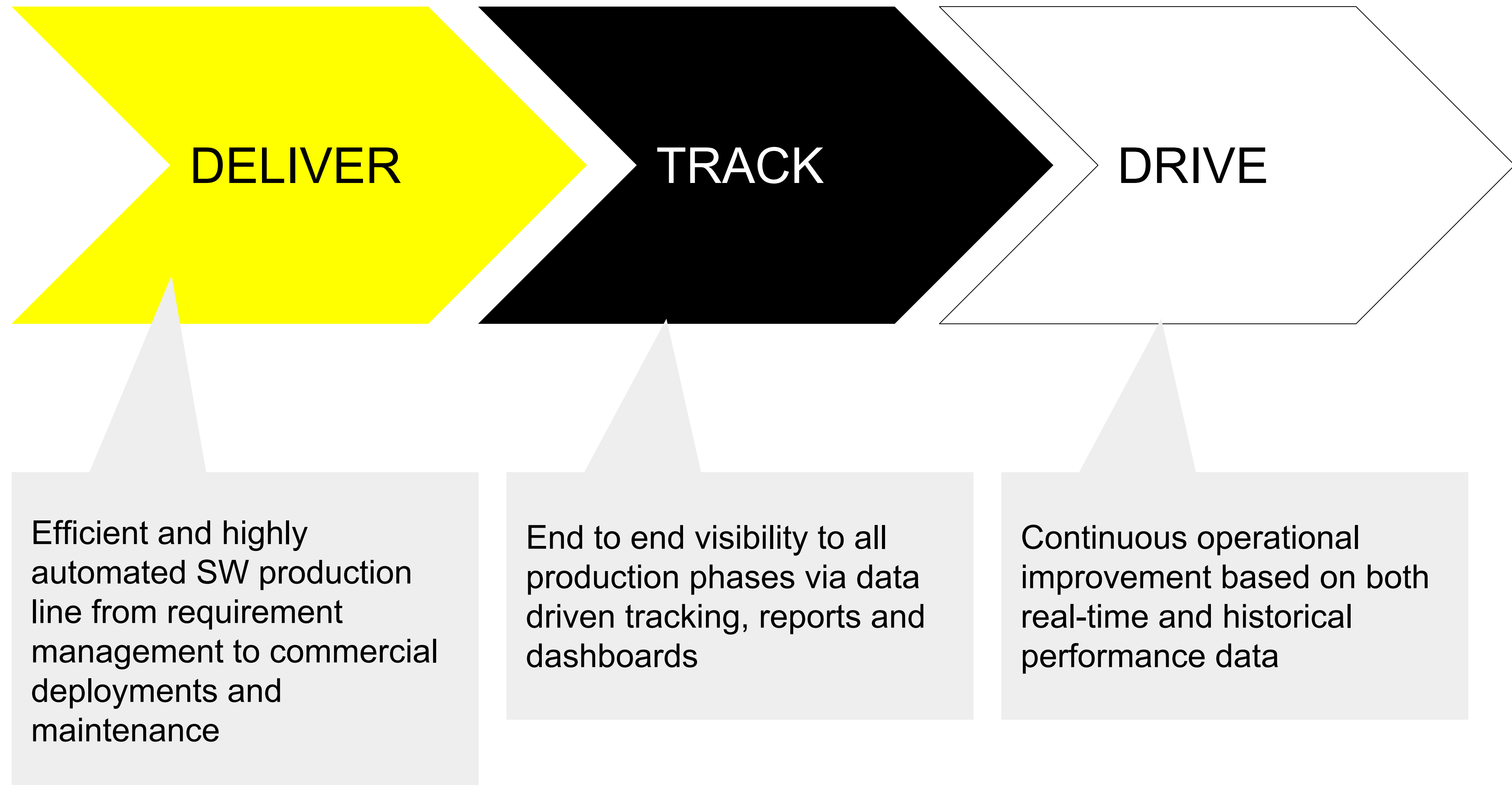


Faster time-to-value

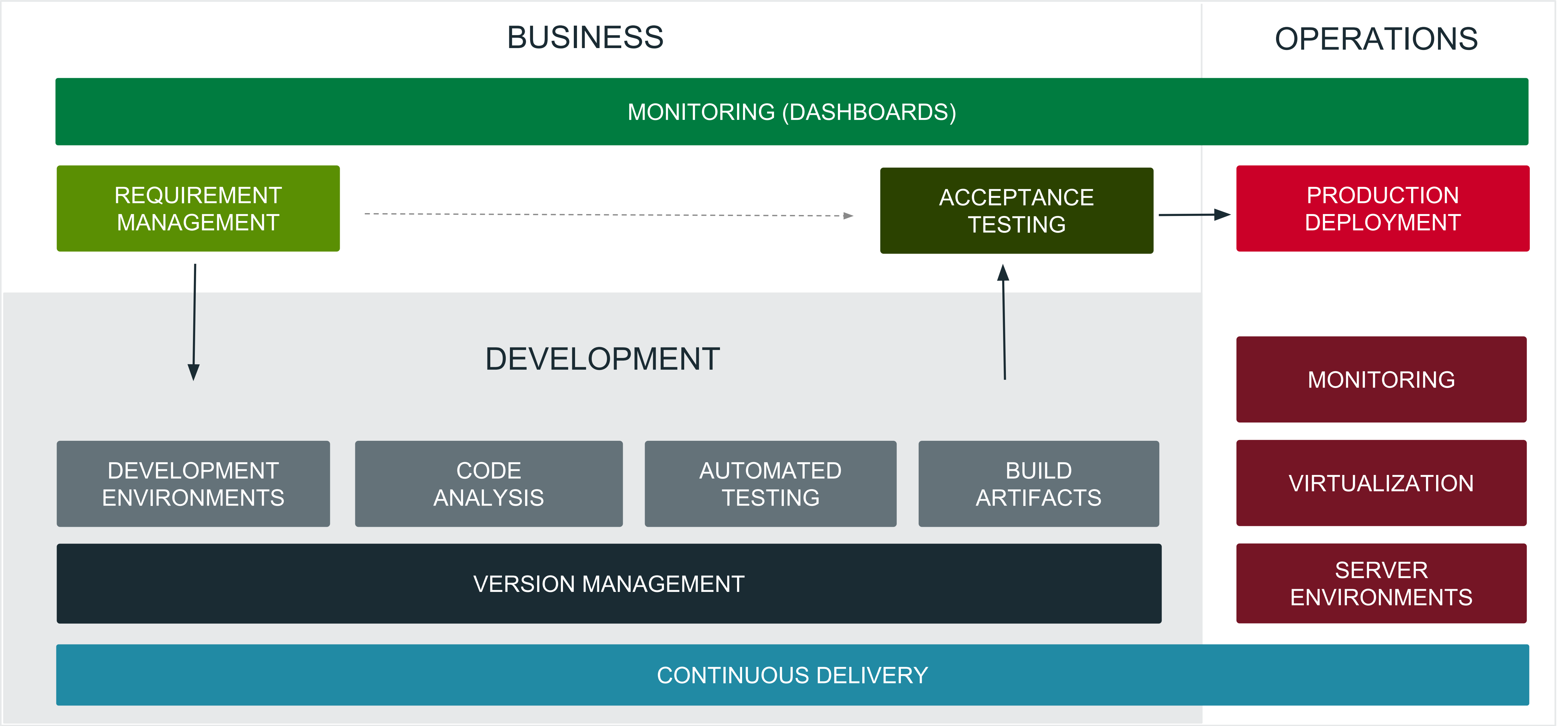
More time to innovate and improve

Improved customer experience

DEVOPS TRANSFORMATION JOURNEY



MODERN SOFTWARE DEVELOPMENT MODEL



SUMMARY

BACKGROUND

- Eficode performed a Devops pre-study with Volvo IT's SCQ team
 - 10 people with varying backgrounds and different job descriptions were interviewed.
 - Background material and SCQ application architecture was reviewed.
- The target was to assess the state of the current Devops implementation and identify areas for improvement
 - Focus on how to increase the product's quality, enhance development and optimize development practices within the Team.
- The evaluation is based on Eficode's maturity model which includes:
 - Devops maturity
 - Automation maturity
 - Test automation maturity

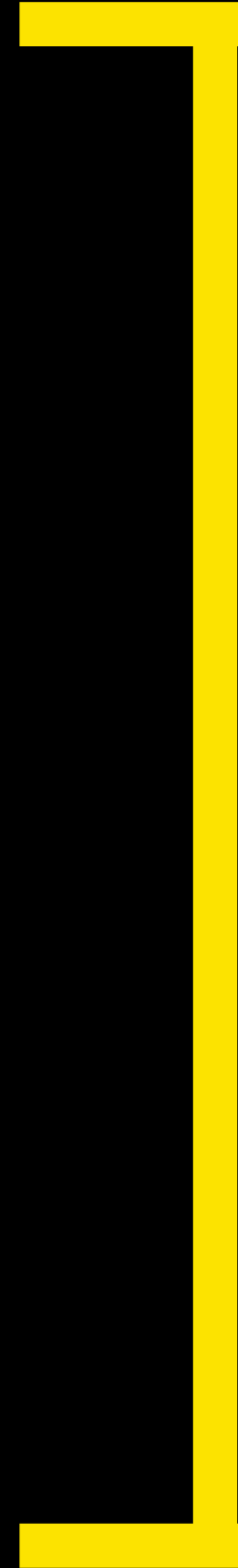
BACKGROUND

- About the Volvo IT SCQ software production environment:
 - The product has a relatively long history and a fairly large code base.
- Architecture is monolithic.
- Serves multiple brands:
 - Renault
 - VCE (Volvo Construction Equipment)
 - NAT (North American Trucks)
 - Serving multiple brands increases application complexity
- Some plans for componentization exist.



VOLVO IT KPIs:

Reduce Lead-time
Increase Quality



DEVOPS MATURITY - 46 / 100



| | 001 | 002 | 003 | 004 |
|-----------------------------------|--|--|--|--|
| LEADERSHIP | Development operations have been separated from the business knowledge. Starting a new development project is laborious. | Starting new development projects is agile, and there are practices in place for steering the project. | New projects can be connected to organization's strategic targets. Starting a new pilot project is easy. | Real-time metrics supporting decision making and tracking the completion of strategic targets are available. |
| ORGANIZATION AND CULTURE | Design, development and quality assurance are separate from each other. Communication is primarily in writing. | Work is conducted in teams but development and quality assurance are separate from each other. | The teams work independently. They have total liability for the development and quality assurance of features. | The teams communicate with each other regularly and work together to improve their practices. Communication with the IT operations is continuous. |
| ENVIRONMENTS AND RELEASE | Products are environment-specific and they are compiled manually. Environments are installed and configured manually. | The system is divided into parts and the compiling environment is known. Some releases are automated. | Environments can be installed and configured automatically. Build and release processes are automated. | Releases may be conducted automatically and continuously. Migration and recovery processes work as expected. |
| BUILDS AND CONTINUOUS INTEGRATION | Product integration is automatic, but configuration and deployment are controlled manually. No artifact or change logs management. | The process starts team-specifically after every change. Tools are shared. Integration does not involve testing. | Integration covers the entire product and it is connected to acceptance testing. Dependencies are known and managed. | Build and integration processes are continuously improved based on collected metrics with aim to speed up the feedback cycle and improve visibility. |
| QUALITY ASSURANCE | Quality assurance is conducted completely by hand and primarily after development. | Unit testing or static code analysis is in place for some parts of the product. | Features visible to the end users are covered with automatic tests. Testers participate in the development process. | Acceptance tests present system requirements clearly and guide the development of the system as much as possible. |
| VISIBILITY AND REPORTING | Reports are made by hand when necessary. | Code integration, unit testing and code analysis are visible to the team. | The status of requirements can be monitored in real time in relation to tests and released features. | Real-time metrics are automatically collected from the product development process and used as a basis for improvement. |
| TECHNOLOGIES AND ARCHITECTURE | Technologies and tools are obsolete or are not fit for current requirements. | Technologies are growing old and the architecture is only partially adaptive or the interfaces are lacking. | Technologies are modern or well supported. The interfaces are well documented and exist for all key functionalities. | The architecture and technologies are optimal and enable reaching business targets efficiently. |

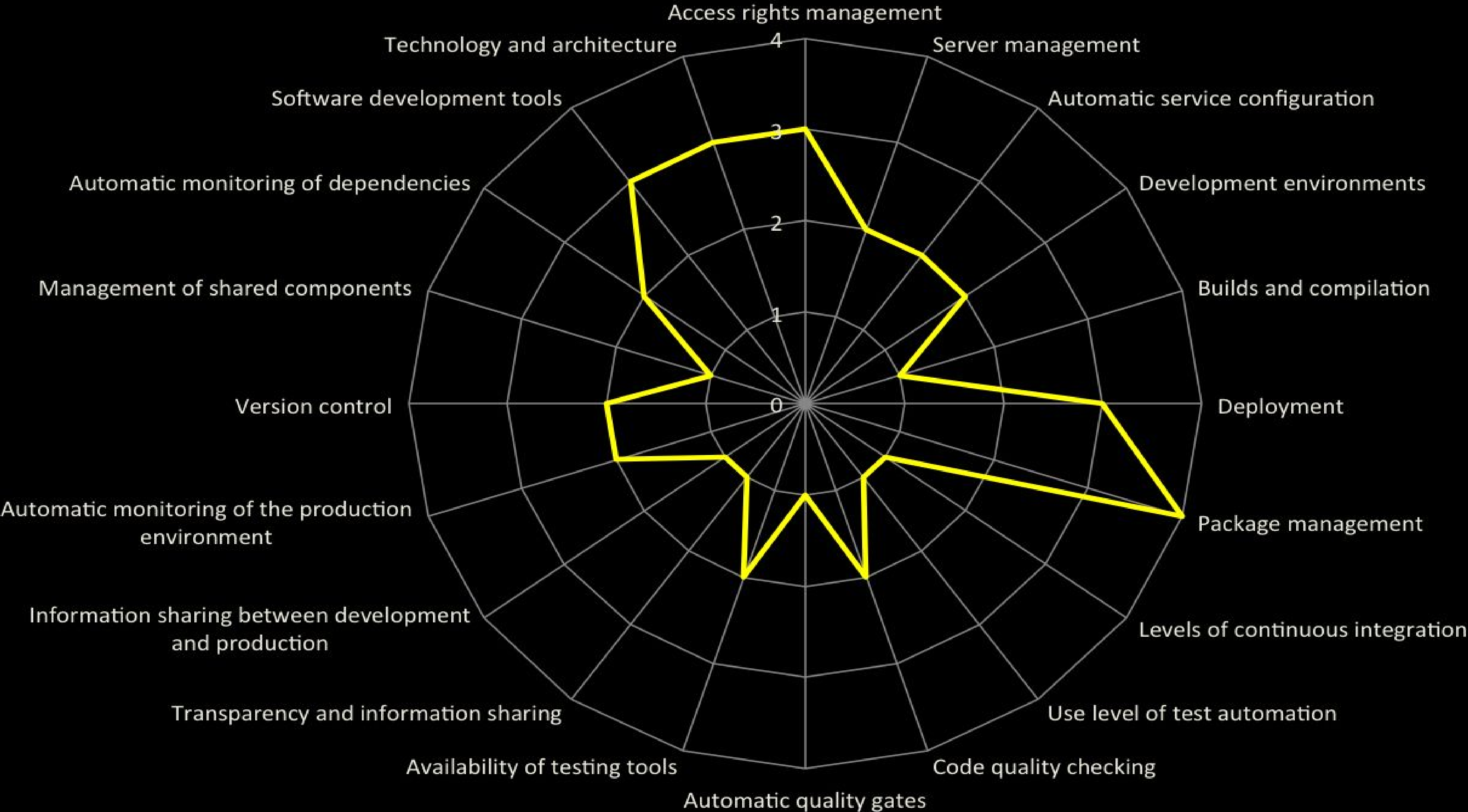
DEVOPS MATURITY - 32 / 100



AVERAGE FOR HEAVY INDUSTRIES

| | 001 | 002 | 003 | 004 |
|-----------------------------------|--|--|--|--|
| LEADERSHIP | Development operations have been separated from the business knowledge. Starting a new development project is laborious. | Starting new development projects is agile, and there are practices in place for steering the project. | New projects can be connected to organization's strategic targets. Starting a new pilot project is easy. | Real-time metrics supporting decision making and tracking the completion of strategic targets are available. |
| ORGANIZATION AND CULTURE | Design, development and quality assurance are separate from each other. Communication is primarily in writing. | Work is conducted in teams but development and quality assurance are separate from each other. | The teams work independently. They have total liability for the development and quality assurance of features. | The teams communicate with each other regularly and work together to improve their practices. Communication with the IT operations is continuous. |
| ENVIRONMENTS AND RELEASE | Products are environment-specific and they are compiled manually. Environments are installed and configured manually. | The system is divided into parts and the compiling environment is known. Some releases are automated. | Environments can be installed and configured automatically. Build and release processes are automated. | Releases may be conducted automatically and continuously. Migration and recovery processes work as expected. |
| BUILDS AND CONTINUOUS INTEGRATION | Product integration is automatic, but configuration and deployment are controlled manually. No artifact or change logs management. | The process starts team-specifically after every change. Tools are shared. Integration does not involve testing. | Integration covers the entire product and it is connected to acceptance testing. Dependencies are known and managed. | Build and integration processes are continuously improved based on collected metrics with aim to speed up the feedback cycle and improve visibility. |
| QUALITY ASSURANCE | Quality assurance is conducted completely by hand and primarily after development. | Unit testing or static code analysis is in place for some parts of the product. | Features visible to the end users are covered with automatic tests. Testers participate in the development process. | Acceptance tests present system requirements clearly and guide the development of the system as much as possible. |
| VISIBILITY AND REPORTING | Reports are made by hand when necessary. | Code integration, unit testing and code analysis are visible to the team. | The status of requirements can be monitored in real time in relation to tests and released features. | Real-time metrics are automatically collected from the product development process and used as a basis for improvement. |
| TECHNOLOGIES AND ARCHITECTURE | Technologies and tools are obsolete or are not fit for current requirements. | Technologies are growing old and the architecture is only partially adaptive or the interfaces are lacking. | Technologies are modern or well supported. The interfaces are well documented and exist for all key functionalities. | The architecture and technologies are optimal and enable reaching business targets efficiently. |

Automation maturity - 48/100



KEY OBSERVATIONS

CURRENT STATE

LEADERSHIP

- Mission is unclear
 - What is the purpose of SCQ? What are we supposed to become?
 - Clear mission is one of the cornerstones of motivation
 - Mission is also important when prioritizing work
- Good level of mastery and autonomy
 - Experienced technical team
 - Business wants you to deliver, but you decide how you do it
 - Very good for motivation

CURRENT STATE

ORGANIZATION & CULTURE

- Operational model reflects organizational model
 - Organization divides teams to devs, testers and analysts
 - Culture encourages responsibility division according to organizational model into devs, testers, analysts
 - Sub-teams within team take responsibilities of part of the flow but not for the entire project
- Culture of fast gains
 - Heavy prioritization of new features and manual testing
 - Low priority on automated tests, refactoring and code reviews
- Agile culture
 - Iteration thinking in place
 - Build increments not products at once

CURRENT STATE

ORGANIZATION & CULTURE

- Information shared mainly in discussions
 - Essentials in daily scrum
 - On-demand discussion in person or utilizing skype
 - Communication is efficient but information only accessible to participants in discussions
- High level of trust in each other - maybe even blind trust
 - No code reviews
 - No pair programming
 - No sharing of responsibility - either he did it or I did - there is no we in doing

CURRENT STATE

PROCESSES

- Segregated flow
 - Requirements definition, development and testing are segregated
 - Always in this order
 - Handovers between flow
- Well implemented scrum
 - Appropriate events and event lengths
 - No overly long meetings
- Definition of done is unclear
 - Not written anywhere

CURRENT STATE

PROCESSES

- Requirements are unclear
 - You have all it takes (appropriate meetings, business participation, 2 analysts and SL/CPM + TL)
 - Maybe testing perspective and details should have more emphasis?
- Improve strategy, make testing a first class citizen
 - Involve testing perspective from the beginning, not only after features are implemented
 - Define high level acceptance test early - clear understanding from beginning on how new feature should work
 - Story should never be considered done if it doesn't have tests

CURRENT STATE

ENVIRONMENTS & RELEASES

- Mutable linux servers managed by HCL
 - Fragile approach by design
- Stable process for server patches
 - Apply patches with Red Hat Satellite
 - Apply first to TEST environment, then make sure app runs like before and continue to QA and PROD
- OpenShift should be production ready later this year
 - Deploy immutable containers whenever you're ready
 - Reliable green-blue zero-downtime deployments
 - You will still have HCL-dependency and potential downtime when version includes DB-migrations

CURRENT STATE

ENVIRONMENTS & RELEASES

- Big and slow releases
 - 4 major releases per year
 - Deadline driven releases
- Inflexible and slow process
 - No ability to release without special arrangement
 - HCL requires a maintenance window
 - 2 weeks UAT before production

CURRENT STATE

BUILDS & CONTINUOUS INTEGRATION

- Requirements for efficient continuous integration:
 - Must have production like environment with production like data
 - Must be able to deploy all new changes and verify quality of new and existing features
 - Standardized, automatic build and deployment processes
 - Quality check needs to be swift - defines feedback loop length
- Current status:
 - Slight differences in TEST, QA and PROD environment
 - Deployments happen when testers are ready to take a new version under test
 - Automated build and deployment processes exist
 - Slow, manual verification of new versions - automation not part of CI

CURRENT STATE

QUALITY ASSURANCE

- Testing is done mostly after development activities are finalized
 - Take the advantage of executing automated tests after each build. Target for fixing problems right after a build fails.
 - Automated test cases should be versioned in the same repository as the application software.
- Testing activities take days or even weeks to be done
 - Product's overall status is unknown for a long time.
 - With last minute corrections the overall status becomes vague.
- Visibility of testing is questionable
 - Acceptance criteria, test cases and test reports should be visible and understandable for everyone.
- Level of automated testing is low
 - Hard to define what kind of overall changes each new change introduces.

CURRENT STATE

QUALITY ASSURANCE

- There should be a documentation that one can easily refer to while creating test cases and testing.
 - Documentation should be accumulated by writing at the refinement session.
- Possible major risks because of human errors.
 - Increasing the level of automation in testing activities would result in giving fast feedback that drive further development. Eg. unwanted changes can be seen faster with unit and automated acceptance tests and corrective actions can be started promptly.
- Testers' main responsibility should be to update automated test sets, not to "test that stuff still works".
 - There seems to be no areas that are "impossible to automate" in SCQ.
 - The value of automated tests suffer from culture, ways of working and tools' utilization.

CURRENT STATE

QUALITY ASSURANCE, VISIBILITY & REPORTING

- Aim to 100% test automation level.
 - First evaluate test automation frameworks and choose the one that fulfill your needs.
 - Create a roadmap for increasing the level of test automation.
 - Execute tests automatically after each deployment in the CI server. They should be an automated quality gate in the CI pipeline.
 - Fully automate all current cases and then start increasing coverage.
 - Start with simple, but necessary tests (“smoke set”) and continue until you have a good feeling about the coverage of automated tests executed after a merge. The next step is not to stop test development activities, instead continue to improve the tests in sync with the product’s development activities.

CURRENT STATE

QUALITY ASSURANCE, VISIBILITY & REPORTING

- Currently, the value of high-quality test reporting is not appreciated.
 - Each test execution must generate a report that accurately reflects the current state of the SCQ version used in testing activities.
 - Test report should be the single source of truth in quality assurance
 - Usually test automation tools provide a good and easy way to produce test reports that are similar and easily comparable.
 - If Your current test framework can't generate reports, try some modern test framework, for example [Robot Framework](#).
- Monitoring relies completely on HCL
 - No visibility for SCQ team.
- No specific data collected about team performance
 - Relies on JIRA built-in reporting functionality.

CURRENT STATE

TECHNOLOGIES & ARCHITECTURE

- Self-hosted git as service provider is not optimal
- The architecture is legacy at this point.
 - This is no surprise for a project that has been on-going this long.
- Refactoring and architectural renovation need to be taken into consideration, but this is recognized in the team.
 - Effective roadmap in place:
 - Plans about branching model in version control.
 - Architecture refactoring to microservice-like.
 - Increasing the level of test automation.
- Improvement suggestions give our vision of the priority order.

WIDER ORGANIZATIONAL ISSUES

- Project thinking in the leadership
 - Most of the time software is a product, not a project
 - Considering a new product or a new version as a “project” is harmful
 - New version (project) might have a new manager
 - Old manager drops responsibility and moves on
 - Even if the manager is the same, pressure towards delivering well in the project scope
 - Encourages towards not dealing with any technical debt - instead just deliver obligatory items fast
- Results are wanted in the way of releases, but quality isn't tracked at the same time through well documented data
 - Encourages deadline driven development and value determination

IMPROVEMENT SUGGESTIONS, BENEFITS AND HIGH-LEVEL ROADMAP

SUGGESTIONS

| # | Title | Difficulty to implement | Description |
|---|----------------------------------|-------------------------|--|
| 1 | Define mission | * | Select a timespan and work together to define a clear mission for SCQ. |
| 2 | Select acceptance test framework | * | Select a modern test automation framework. Test cases and report should be clear enough for business people to read and comprehend. |
| 3 | Automated acceptance tests | *** | Work together as a team to implement automated acceptance tests. Start with critical business flows. Reuse current automated tests and HP Quality Center cases to build a core for automated acceptance tests. Abandon HP Quality Center and legacy geb tests. To unify versioning, make sure acceptance tests are in the same git repository with code. |
| 4 | Core unit tests | *** | Testing algorithms and different configuration combinations is not feasible in acceptance tests. Increase unit test level to cover all functions with important business logic in the application. Start using SonarQube to systematically track unit test coverage. |
| 5 | Definition of Done | * | Remove ready for testing column and start implementing testing along with code. When defining subtasks for a story, define also subtasks for unit and acceptance test implementation. Define and document Definition of Done so that it includes also test creation. |
| 6 | Git workflow | ** | Define and document a proper git branching strategy to support new Definition of Done and proper workflow. Start working together as development and testing in feature branches until the feature and both unit and acceptance tests are implemented and only then merge feature to master. Apply a code review process to merging into master. |

SUGGESTIONS

| # | Title | Difficulty to implement | Description |
|----|-----------------------------|-------------------------|--|
| 7 | Continuous integration | * | Trigger unit tests, build and deployment to test environment on every merge to master. Trigger automatic acceptance tests on every deployment to verify quality of new and existing features. |
| 8 | Propose new release process | ** | Whenever new EPICs are completed and tests pass in test environment, deploy to QA and let business know you're ready for release any time they want. Also pass acceptance test reports for them to review. Ideally, you should be able to release into production after every sprint. However, if UAT is required, you can have a column "waiting for UAT" for ready stuff and business can test and approve whenever they want. |
| 9 | Componentization plan | * | Establish an understanding of individually deployable components in SCQ. Make sure no components are both services and clients. |
| 10 | Componentization execution | *** | Conduct componentization. Refactor SCQ into separately deployable components. |
| 11 | Transition into OpenShift | ** | Put components into docker containers and deploy them to OpenShift when it's officially production ready. |
| 12 | Build container monitoring | ** | Start monitoring your containers and take end-to-end product responsibility. Use a modern container monitoring tool, like for example Prometheus. |

BENEFITS

| # | Title | Description |
|---|----------------------------------|--|
| 1 | Define mission | Mission is crucial for Team's motivation. It is essential to know what a Team is aiming for in order to unify goals and work towards a common mission. |
| 2 | Select acceptance test framework | Creating automatic acceptance tests is a Team effort. Taking a moment to evaluate and deciding the tools together is the first step in getting everyone on board on test automation. |
| 3 | Automated acceptance tests | Currently it takes some effort to get an overall picture of SCQ quality. Furthermore, development and testing are conducted sequentially and developers need to wait for a long time before they get a verification of produced quality. Automatic acceptance tests allow testers and developers to put effort on the same feature more in parallel and reduce the regression feedback cycle for developers. Automatic acceptance tests' reports also act as a real time quality status for the product. Manual testing also prevents effective use of continuous integration, since testing takes days or even weeks and requires code version to be frozen for this period. With automatic tests every new software version can be deployed and tested automatically and in reduced time. Cost per testing iteration goes down compared to manual testing. |
| 4 | Core unit tests | Good unit test coverage gives the Team confidence to proceed post haste. Unit tests that are executed every time a code commit is done ensure that the low level code components work as expected and don't produce surprising results. Unit tests are also cheap to run and catching bugs this early means that they are easier and cheaper to fix compared to later stages of the project. |
| 5 | Definition of Done | Having a written Definition of Done brings clarity into software delivery process. If Definition of Done is not written anywhere, people may have different opinions or perceptions of whether or not a task is done. Including test case development in Definition of Done is the first step towards a less siloed team culture, where testers and developers mostly work together on features instead of taking turns on them. |
| 6 | Git workflow | Git workflow has a big impact on software production process. Reshaping your git workflow allows you to support the new Definition of Done. Workflow also defines what a change is. If you define a change to be every commit, then processing every change in CI can be quite chaotic. Defining a change to be every feature merge to master drastically reduces the pace of continuous integration so that building, deploying and testing every change becomes feasible. Furthermore, it is the optimal moment for code reviews, if you ever wish to apply review process to every change. |

BENEFITS

| # | Title | Description |
|----|-----------------------------|--|
| 7 | Continuous integration | With a true Continuous integration environment you gain the possibility to see the basic overall status of the software after someone has done a new software update. The new software has been verified that it's buildable, basic tests have verified that it's not broken, the software has been updated to a dedicated test environment and automated acceptance tests verify that the software's quality hasn't decreased. |
| 8 | Propose new release process | With the renewed release process the Team has the possibility to provide business updated application software after each EPIC has been finalized. The software quality has been verified with acceptance tests in the Test environment and exactly same application software version has been provided to the QA environment for the business for their review. Additionally business has the possibility and power to accept the new application software and get it released to the Production environment. |
| 9 | Componentization plan | In order to proceed with the transformation to a microservice model the application's components have to be studied and their individual role understood. Each component has a dedicated role, some are services and some are client, but none should be both. |
| 10 | Componentization execution | Components enable easier management of the software. Logic is compartmentalized into logical entities and managed independently. Changes to production are lighter when components can be changed independently instead of a single monolithic software. Ripple effect of changes to other components is reduced considerably. |
| 11 | Transition into OpenShift | Put components into docker containers and deploy them to OpenShift when it's officially production ready. Transition into OpenShift supports proper separation of responsibilities. Supplier should be responsible of the platform and you should be responsible of deployment and uptime of your application. End-to-end responsibility is in the core of Devops teams and gives you the freedom of deploying new versions of components in a reliable manner without any need for supplier intervention. OpenShift is built to support this, where supplier runs the platform and you deploy your application via an automatic interface. Also, this will probably cause heavy cost reduction in operations. |
| 12 | Build container monitoring | Even though OpenShift can automatically relaunch your application when it fails, monitoring and alerting are essential for you to know that something is wrong so that you can start investigating the root cause of issues and apply fixes to your application. |

HIGH LEVEL ROADMAP

2017

2018

09

10

...

01

02

...

11

12

Phase 1

- Define mission
- Select acceptance test framework
- Automated acceptance tests
- Core unit tests
- Definition of Done
- Git workflow

Phase 2

- Continuous integration
- Propose new release process
- Componentization plan
- Componentization execution

Phase 3

- Transition into OpenShift
- Build container monitoring

APPENDIX 1: AUTOMATION MATURITY MATRIX

| Environments and builds | | | | |
|---------------------------------|--|---|--|---|
| Access rights management | Every tool has separate user management or actual access rights management is not in use. | Each tool has a separate user management, but persons in charge are documented or the channel for requests is common. | Shared tools are connected to the company's centralized access management. Tool specific access rights are handled with group management. The system enables centralized creation of non-personal integration accounts. | Group management is project specific and projects themselves can manage their groups. The system enables creation of light R&D accounts in under an hour. |
| Server management | Servers are requested by mail or a tool several weeks or months earlier. Access management is manual. | Acquiring servers is done by a tool, but getting them takes weeks. Acquisition is centered to specific persons. There's a process for documenting environments. | Servers can be requested instantly with an electronic tool. | Servers acquired for development work are destroyed automatically. Access management on development is connected to group management and is connected to other development tools. |
| Automatic service configuration | External services, such as firewalls, load balancers and data warehouses as well as servers are configured manually. | Servers are configured manually, but external services automatically | Services are mainly configured automatically | All components of the service are automated. The whole infrastructure of the service is described as code. |
| Development environments | Developers can choose their own tools, if they are free. Everyone installs their own tools and are responsible for their functioning themselves. | Every developer is given a specific tool set. The company pays for commercial tools, if they are used. | Every developer is given a specific tool set. Installation is documented and IT support gives full support for the installation. Based on their needs, developers can also use their own tools, if they're free. The company pays for commercial tools, if there's basis on purchasing the tool. | Creation of development environments is automated. Building a new environment to an empty machine is easy, fast and reliable. |

| Build and continuous integration | | | | |
|---|--|--|---|--|
| Builds and compilation | <i>Version control and installation is not connected without manual steps.</i> | A compilation and or quality assurance is automatically started from version control, but not the packaging or the installation. | Distributable packages are developed automatically from version control to the artifact storage. | Version control changes passing quality assurance are automatically taken to installation. |
| Deployment | Installation is done manually and few people are able to do them. | Installation is documented and documentation is updated regularly. | <i>Application update is automated and both utility program and configuration needs are documented.</i> | Application and the infrastructure it needs are completely automated. |
| Package management | No package management or differing conventions. | Packages are stored in version control. | Package management has a separate storage system, where packages are stored versioned. Requirement relations are documented. | <i>Packages are handled with system that is specialized to package management, which can be used to make queries about requirement relations.</i> |
| Levels of continuous integration | <i>No continuous integration or loose conventions.</i> | Continuous integration is team specific and doesn't cover the whole product pipe. | Results of integration process of teams is available for other teams, but integration of the next level is not started automatically. | Team level integration, when successful, starts integration of the next level automatically and relevant teams get information about the integration results related to components they are responsible. |

| Quality assurance | | | | |
|-------------------------------|--|---|--|--|
| Use level of test automation | No automated tests (manual/no tests) | Smoke tests / regression tests for main features. | Large part of the application is tested. | Over 85% test coverage. |
| Code quality checking | No special code quality monitoring. | One of three in use: Static analysis, pair programming, code review | Two of three in use: Static analysis, pair programming, code review | All three in use: Static analysis, pair programming, code review |
| Automatic quality gates | No automatic quality gates. | Qualities are in the process definitions: code reviews done by people and reports done by testers act as quality gates. Coding style conventions are defined. | There's a review tool for code review and test automation produces automatic reports. | In addition to automated tests and possible code review, software is checked with static analysis with every version control commit. |
| Availability of testing tools | All testing is done manually by testers or automatic tests can't be run locally. | Smallish part of tests can be run on developers' machines or tests requiring external dependencies are not separated. | Most tests can be run on developers' machines. Tests are organized so that tests requiring externals dependencies can be easily left not run. A test environment is available for all external dependencies. | All tests can be run on developers' machines. External dependencies are either replaced with imitations or can be run locally. |

| Visibility and reporting | | | | |
|--|--|---|--|--|
| Transparency and information sharing | Information about development and production is scattered and formed manually. | Some teams produce information about their actions automatically. Information might be centralized, but is not available for everyone and no combined analysis is done. | Fault situations of the most critical environments are shown in the workspaces of the responsible teams. Reports of the development process is automatically generated at least monthly. | The status of all systems is available for both development and productions teams and their managers on all organization levels. Development produces portfolio level information to upper management and it is easily available and almost real time. |
| Information sharing between development and production | Development and production is separated. Production can't access development documentation and developers can't access metrics produced in production. | Production and development is separate, but reports are shared as needed. A process exists for sharing information. | A part of tools used by production and development are common and information is shared between the team regularly. | Development team is able to follow problems occurring in production at any time. Production can forward tickets they have accumulated to development, if the problem can't be fixed without the help of the development team. Production and development can communicate fluently or the teams are combined. |
| Automatic monitoring of the production environment | The status of the service is not automatically monitored. | Some components of the service are monitored automatically or manual monitoring (e.g. disk space) has a well documented process. | All components of the service are monitored automatically, but total availability is not calculated or reacting to fault situations is not done automatically. | The metrics of different components and servers of the service are constantly monitored and they leave history information. Availability of the distributed infrastructure is calculated. Reacting to fault situations is done automatically. |
| Version control | Version control is not used or the use is sporadic. | Version control has a documented process, but the process is followed randomly or it does not serve good practises. | Version control process is followed extensively and procedures cover special situations. | Developers work with systematic version control. Problems caused by inadequate use of version control practically don't appear. Versions deployed to production can be easily linked back to version control. |

| Technology and architecture | | | | |
|--------------------------------------|---|--|--|--|
| Management of shared components | Sharing of components relies on tacit knowledge. | Information about shared components is maintained in documentation. | Installation packages of shared components are usually available. | Copies of the components can be installed from a centralized place and notification of their updates are available. |
| Automatic monitoring of dependencies | Dependencies are not followed structurally. | Dependencies are managed with manual documentation. | Component dependencies are followed automatically, but centralized dependency information is not available or in use for those developing internal dependencies. | Component dependencies are followed automatically. Information is available for other teams as well. The information contains version and other information of the dependencies. |
| Software development tools | Every team is responsible for their own development tools. | Teams are responsible for their own tools, but the tool management is centralised. Backups and monitoring works in device level. | Some of the tools are centralised, but they do not offer different workflows for different teams or tools are managed manually. | All shared tools are centralised to common environment. Teams can manage automated project creation inside the team. Tools enable different workflows for each team. |
| Technology and architecture | Technology is old and/or does not meet the requirements of the architecture in scalability, integration and security aspects. | Technology is about to be too old and architecture is flexible only in some parts or interfaces are inadequate. | Technology is quite new and well supported. Architecture is separated and all frontend functionality can be done also using interfaces. | Technology is new or well supported. System structure is lead as a whole and is divided into appropriate microservices. |

APPENDIX 2: TEST AUTOMATION MATURITY MATRIX

| Test Automation Maturity | | | | |
|--------------------------|------------------------|---|---|---|
| Unit Testing | No unit tests | Unit tests are executed locally on the developer's machine. | Unit tests are executed automatically on the CI-server. Failed tests prevent from releasing. There is something to be desired in respect of test coverage. | Unit tests are fully automated and their test coverage is over 85%. |
| Integration Testing | No integration tests | Integration tests are executed manually. | Integration tests are executed automatically on the CI-server. Failed tests prevent from releasing. There is something to be desired in respect of test coverage. | Integration tests are fully automated and their test coverage is over 85%. |
| System Testing | No system tests | System tests are executed manually by the tester. | System tests are executed automatically on the CI-server. Failed tests prevent from releasing. There is something to be desired in respect of test coverage. | System testing is fully automated and its test coverage is over 85%. |
| Performance Testing | No performance testing | Performance tests are executed manually. | Performance tests are executed automatically on the CI-server. Failed tests prevent from releasing. There is something to be desired in respect of test coverage. | Performance testing is fully automated and it simulates well user load and actions. |
| Security Testing | No security testing | Security tests are executed manually. | Security tests are executed automatically on the CI-server. Failed tests prevent from releasing. There is something to be desired in respect of test coverage. | Security tests are fully automated and their test coverage is on a good level. |

THANK YOU!

Mikko Drocan

mikko.drocan@eficode.com

Markus Suonto

markus.suonto@eficode.com

