



# PRODUCT ENGINEERING MINDSET - WORKSHOP

Balaji Thiruvengadam

## DAY 8 - AGENDA

---

Recap of day 7

---

.NET code analyser

---

Continuous Integration, Deployment  
and Delivery

---

Product Engineering into Agile

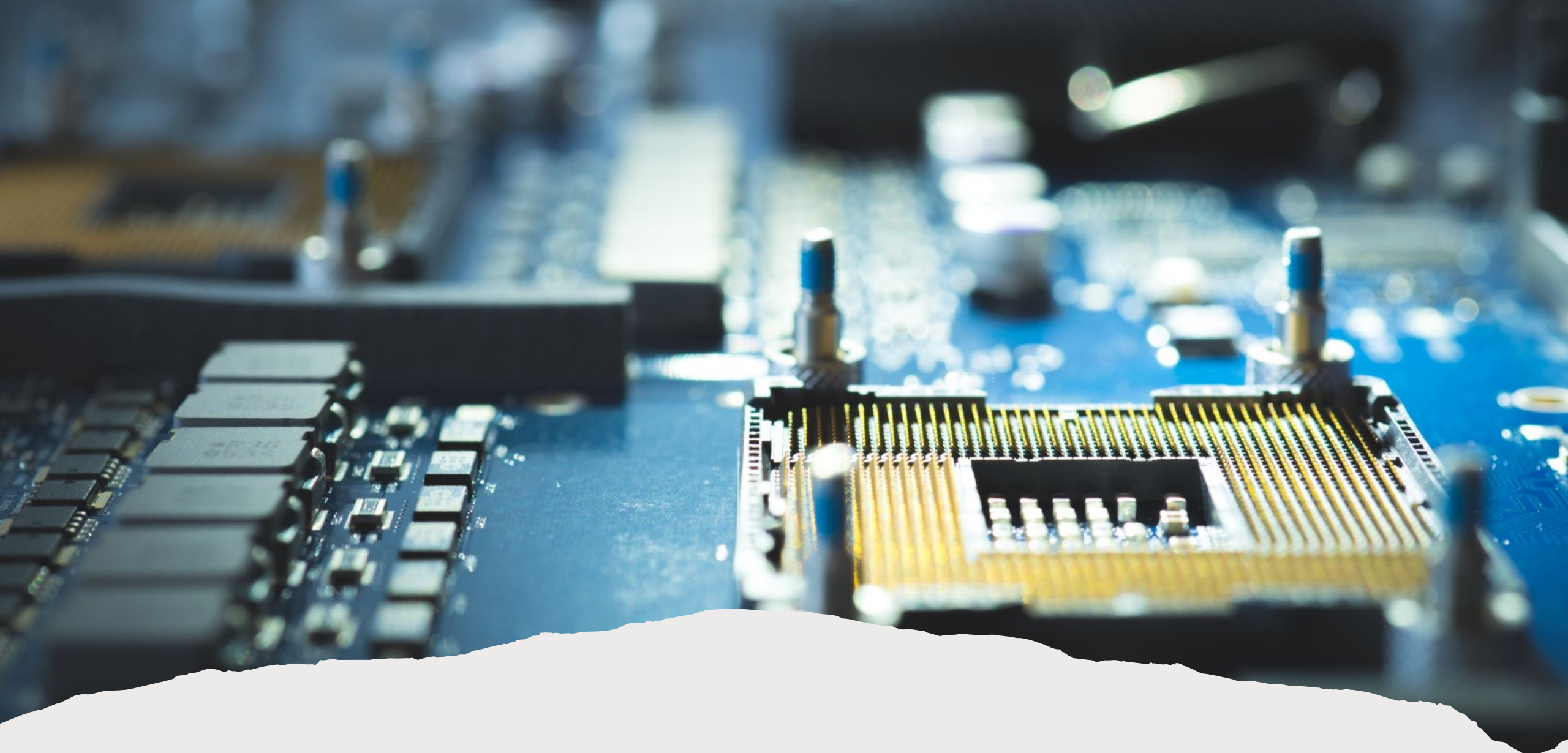
---

Build Good user story

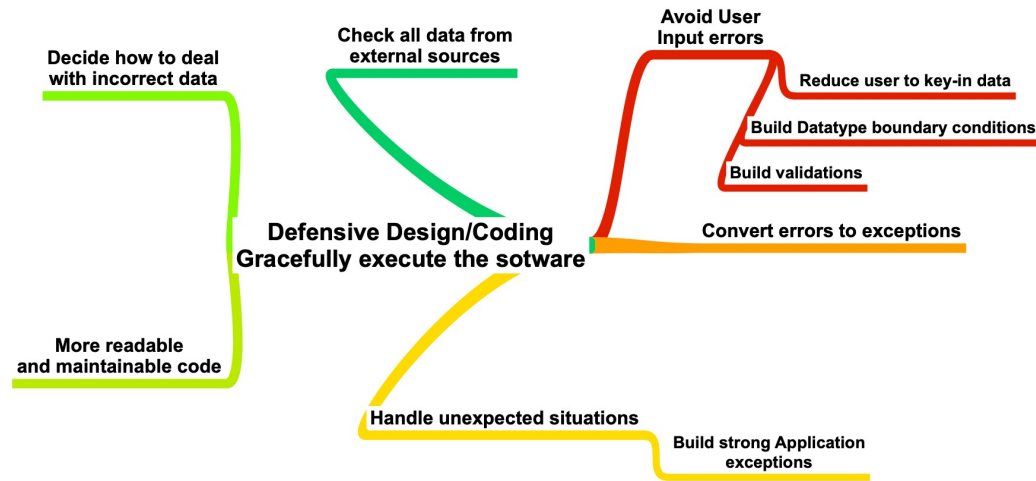
---

Replay of all 8 days





PRODUCT ENGINEERING – RECAP DAY7



	Low Business Value / Low Complexity	High Business Value / High Complexity
High Change Rate	Refactor Units	Refactor Units & Design
Low Change Rate	Don't touch it	Use a Facade

DEFENSIVE CODING & CONTINUOUS REFACTOR

# WHY CODE ANALYSIS?

- Continuously seeing an increase in code changes without understanding what those changes are doing, can quickly lead to possible bugs, application failures and vulnerabilities. These are risks at various levels – corporate data exposure risks, business continuity risks, delays in releases, and a lot more.
- Human reviews are hard to catch all the problems, hence automating the code analysis is an important aspect of product engineering
- At the team, or organizational level, you can define a set of policies and rules that should apply to all the code your team(s) write and commit to your code repositories. This means that you can also centrally track and approve or reject changes that will fly into your production workloads, depending on the quality report of a build.

# CODE ANALYSIS – DEVELOPMENT VS BUILD

## *Development*

- ***Inspect code, development-time.*** Using build-time code analysis in Visual Studio (or other preferred tool), we enable developers to quickly understand what rules are being broken. This enables them to fix code earlier in the development lifecycle, and we can avoid builds that fail later.
- Detect code ***vulnerabilities*** or code that may breach ***corporate policies*** or general ***best practices***.
- Increase the efficiency, and strengthen the confidence of quality code produced.

## *Build*

- ***NuGet*** packaged analyzers are the easiest, and they will automatically run as your project builds on the build agents.
- Enforce code quality and best practices across developers.
- Reject code changes that introduce vulnerabilities, or code that introduces policy-breaking changes.
- Automate everything; When a build encounters a code quality error, you can immediately fail the build, send alerts, or apply any other actions you and your team needs.
- Help your operations-team to get insights into the quality of the delivery, before a major release.

# OPTIONS TO RUN CODE ANALYSIS

## *Code analysis from your development box*

- If the analyzers come with an *extension* for Visual Studio or Visual Studio Code, you can enable the developers to first-hand get insights into the coding practices that needs to be followed.
- This does not come with a requirement to add any dependencies to your solutions and projects. Instead, the code analysis tools run from your developer box when you want them to. Most often when you build your code, or if it is a Linter, it runs as you type.

## *Code analysis using NuGet analyzers*

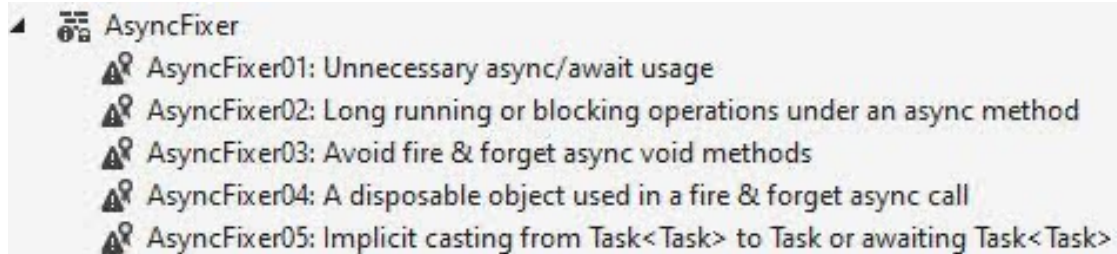
- With NuGet packaged analyzers, we can run them both a local developer boxes during local build, as well as in our build pipelines.
- This creates a dependency on the NuGet analyzers in your projects, but comes with the convenience that regardless if you build locally or on a build agent, you get the same analyzers applied to the code base.
- The entire team benefits here, as all code is passed through the same checks.

## *Code analysis with build pipelines / releases*

- With build pipelines we get a lot of automation. Having automation is great for this part of our work as well.
- In my build pipelines I can easily ensure, and enforce, that a certain level of code correctness and code analysis runs and comes out with good quality before a build is flagged as successful.
- We can run a lot more tools here than the NuGet analyzers, of course. We can run third party Application Vulnerability Scanners, and a lot more great tools to help us stay safer as we deploy more code.



# FAVOURITE CODE ANALYSIS TOOLS FOR .NET CORE



Available as:

- [AsyncFixer](#) (NuGet)
- [AsyncFixer](#) (Visual Studio Marketplace)

## *AsyncFixer*

When you have complex solutions, or even simple solutions that depend on code outside of your comfort zone or knowledge, this tool is great for understanding where you could have potential thread blockers and other async-related issues.

Things it checks:

- Unnecessary async/await methods.
- Using long-running operations under Async methods.
- Fire-and-forget patterns.
- Implicit downcasting from Task<T> to Type



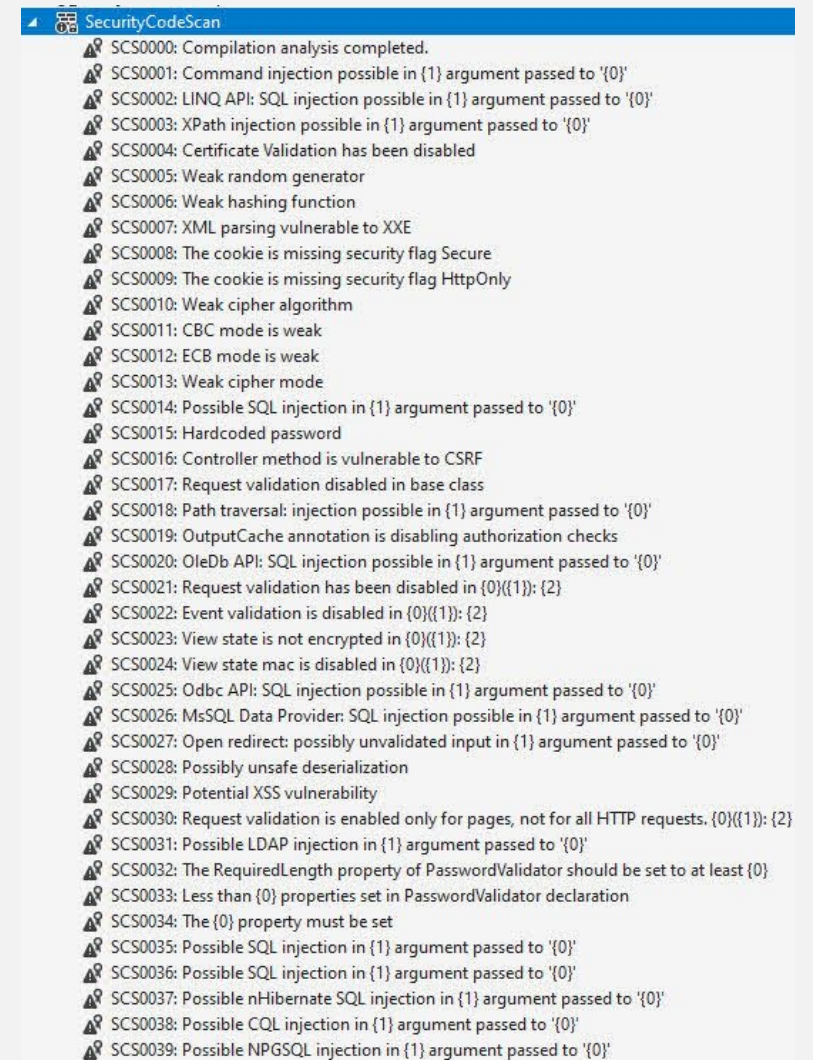
# FAVOURITE CODE ANALYSIS TOOLS FOR .NET CORE

## *SecurityCodeScan*

- The SecurityCodeScan is a Roslyn analyzer that specifically checks for vulnerabilities in your C# code.
- It checks around 40+ code rules right now:

Available as:

- [SecurityCodeScan](#) (NuGet)
- [Security Code Scan](#) (Visual Studio Marketplace)



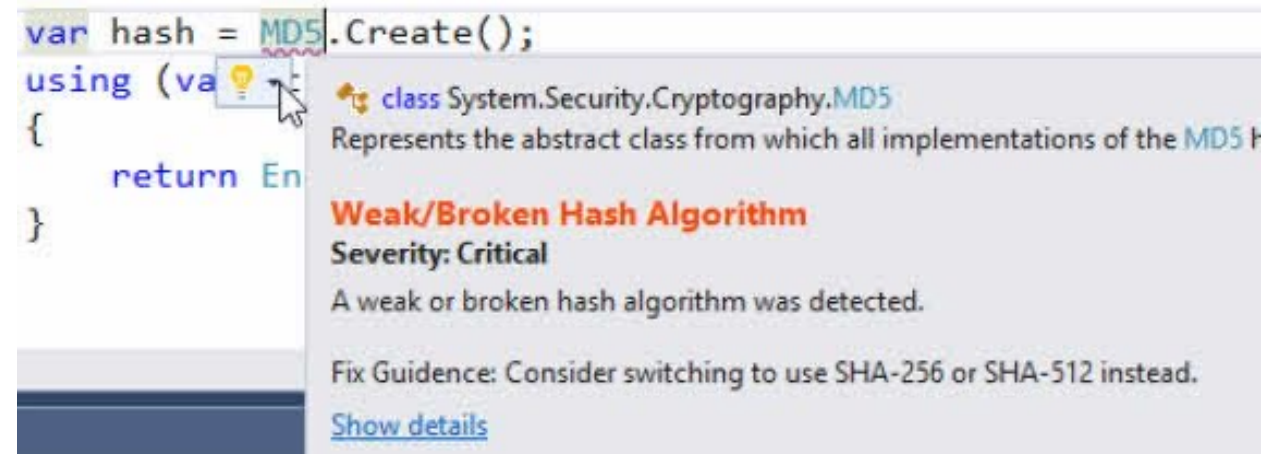
The screenshot shows the SecurityCodeScan application window with a list of 39 security rules. Each rule is preceded by a warning icon (a triangle with an exclamation mark). The rules are as follows:

- SCS0000: Compilation analysis completed.
- SCS0001: Command injection possible in {1} argument passed to '{0}'
- SCS0002: LINQ API: SQL injection possible in {1} argument passed to '{0}'
- SCS0003: XPath injection possible in {1} argument passed to '{0}'
- SCS0004: Certificate Validation has been disabled
- SCS0005: Weak random generator
- SCS0006: Weak hashing function
- SCS0007: XML parsing vulnerable to XXE
- SCS0008: The cookie is missing security flag Secure
- SCS0009: The cookie is missing security flag HttpOnly
- SCS0010: Weak cipher algorithm
- SCS0011: CBC mode is weak
- SCS0012: ECB mode is weak
- SCS0013: Weak cipher mode
- SCS0014: Possible SQL injection in {1} argument passed to '{0}'
- SCS0015: Hardcoded password
- SCS0016: Controller method is vulnerable to CSRF
- SCS0017: Request validation disabled in base class
- SCS0018: Path traversal: injection possible in {1} argument passed to '{0}'
- SCS0019: OutputCache annotation is disabling authorization checks
- SCS0020: OleDb API: SQL injection possible in {1} argument passed to '{0}'
- SCS0021: Request validation has been disabled in {0}({1}); {2}
- SCS0022: Event validation is disabled in {0}({1}); {2}
- SCS0023: View state is not encrypted in {0}({1}); {2}
- SCS0024: View state mac is disabled in {0}({1}); {2}
- SCS0025: Odbc API: SQL injection possible in {1} argument passed to '{0}'
- SCS0026: MySQL Data Provider: SQL injection possible in {1} argument passed to '{0}'
- SCS0027: Open redirect: possibly unvalidated input in {1} argument passed to '{0}'
- SCS0028: Possibly unsafe deserialization
- SCS0029: Potential XSS vulnerability
- SCS0030: Request validation is enabled only for pages, not for all HTTP requests. {0}({1}); {2}
- SCS0031: Possible LDAP injection in {1} argument passed to '{0}'
- SCS0032: The RequiredLength property of PasswordValidator should be set to at least {0}
- SCS0033: Less than {0} properties set in PasswordValidator declaration
- SCS0034: The {0} property must be set
- SCS0035: Possible SQL injection in {1} argument passed to '{0}'
- SCS0036: Possible SQL injection in {1} argument passed to '{0}'
- SCS0037: Possible nHibernate SQL injection in {1} argument passed to '{0}'
- SCS0038: Possible CQL injection in {1} argument passed to '{0}'
- SCS0039: Possible NPGSQL injection in {1} argument passed to '{0}'

# FAVOURITE CODE ANALYSIS TOOLS FOR .NET CORE

## *DevSkim*

- Microsoft has a great tool called DevSkim, which is basically a Linter that helps you with security-related coding practices.
- There is a repository under [Microsoft/DevSkim](https://github.com/microsoft/devskim) on GitHub, where most of the information is available or linked.



Available as:

- [Microsoft.CST.DevSkim](#) (NuGet)
- [DevSkim](#) for VS 2019 (Visual Studio Marketplace)
- [DevSkim](#) for VS Code (Visual Studio Code marketplace)

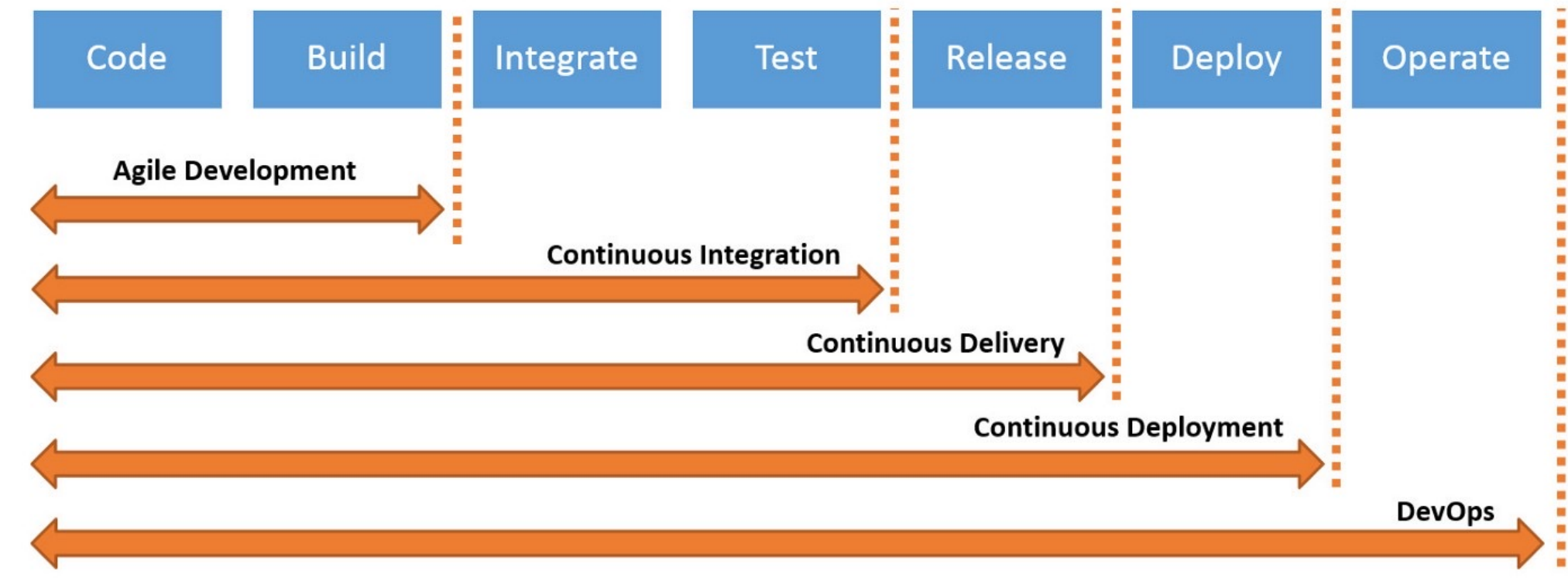
# FEW MORE TOOLS TO LOOK

- Roslynator
- SonarLint and SonarAnalyzer
- RefactoringEssentials



# CONTINUOUS INTEGRATION, DELIVERY & DEPLOYMENT

- CI / CD processes are the inevitable items of medium and large scaled software projects. They are also essential concepts in agile and devops cultures.





# CONTINUOUS INTEGRATION

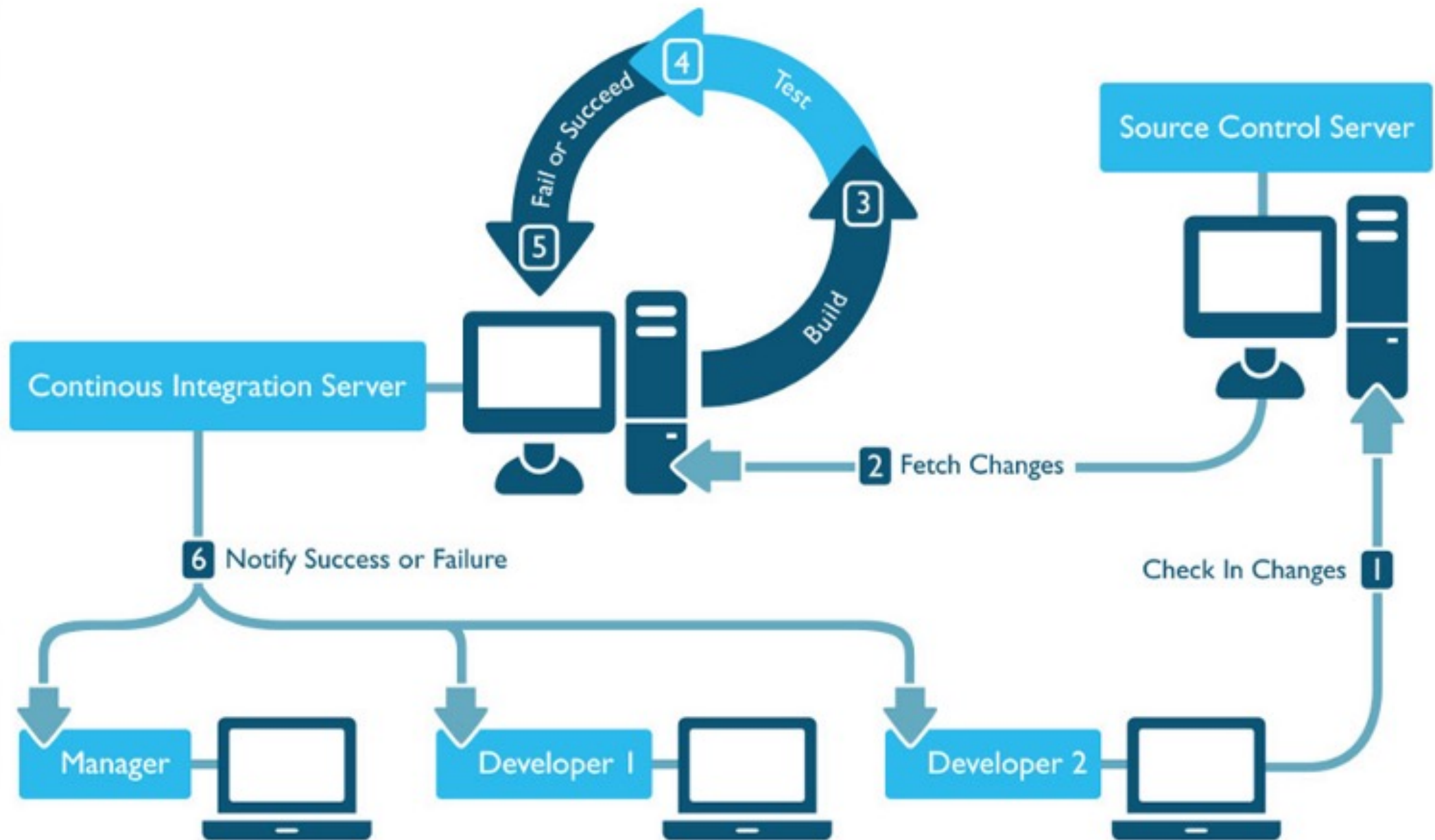
*Continuous Integration* is the system that aims to continuously integrate the code generated by the development team with the codebase. The integration process must be automated by some specific tools that uses scripts and processes as frequently as needed. This way the compatibility of the new code is being tested automatically and this also supports the integration errors being found earlier. In addition to that, unit tests or system level tests can be done automatically **without any human effort**. Running tests frequently without any effort ensures that the system is reliable and stable, resulting in high quality products. In summary, benefits of Continuous Integration are:

- ★ errors are found and fixed earlier
- ★ monitoring and measuring the health of the system is easier
- ★ testing is done over working software any time, not based on assumptions,
- ★ repeated human effort is decreased
- ★ working software can be delivered anytime or in a very short time
- ★ transparency is built
- ★ trust is established inside and outside the team.



# CONTINUOUS DELIVERY & DEPLOYMENT

- Releasing of the package that is generated in CI phase, to the environment usually called “development”, “test”, “staging”, “pre-production”. This releasing work can be scheduled according to business requirements like daily, weekly. Quality assurance (QA) tests are run in this phase. Deployment (Production release) is done *manually* in Continuous Delivery pipelines.
- If the production releasing is also automated then this process is called “*Continuous Deployment*.” It means user acceptance tests (UATs) are automated.
- Both Continuous Delivery and Continuous Deployment save us the trouble of *Release-Day-Pain*. Because we already have a running, versioned, tested deployments in our environments.



## Source Control



1. Branch

2. Push

*Github, Gitlab*

## Build Server



3. Build and test

4. Nuget package

*Jenkins, TeamCity*

## Automated Deployment



5. Deploy Internally

6. Deploy Production

*Octopus*

## Monitor



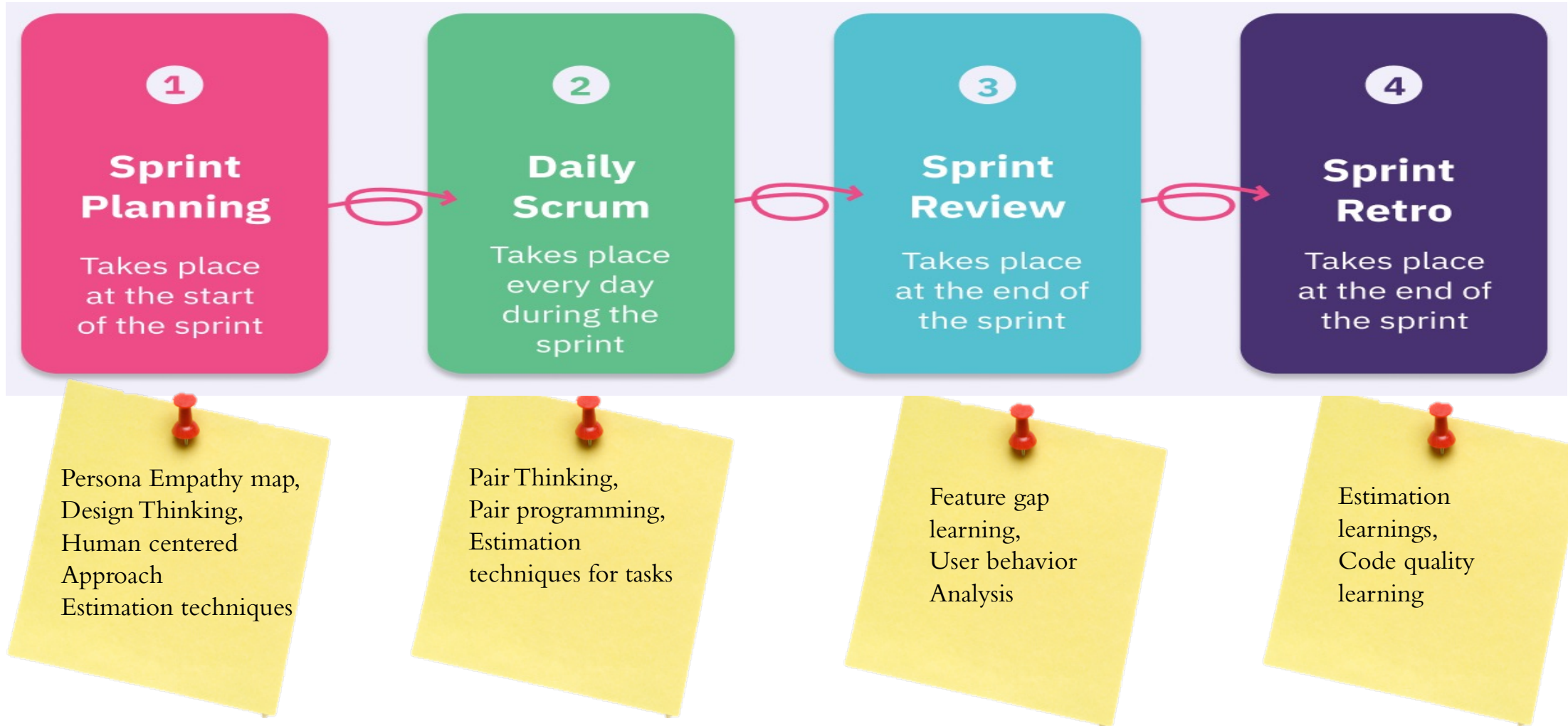
Crash reports

Performance metrics

*Stackify, dotTrace, AQTime*

# CI/CD TOOLS

# HOW TO BRING PRODUCT ENGINEERING PRACTICES INTO AGILE CEREMONIES

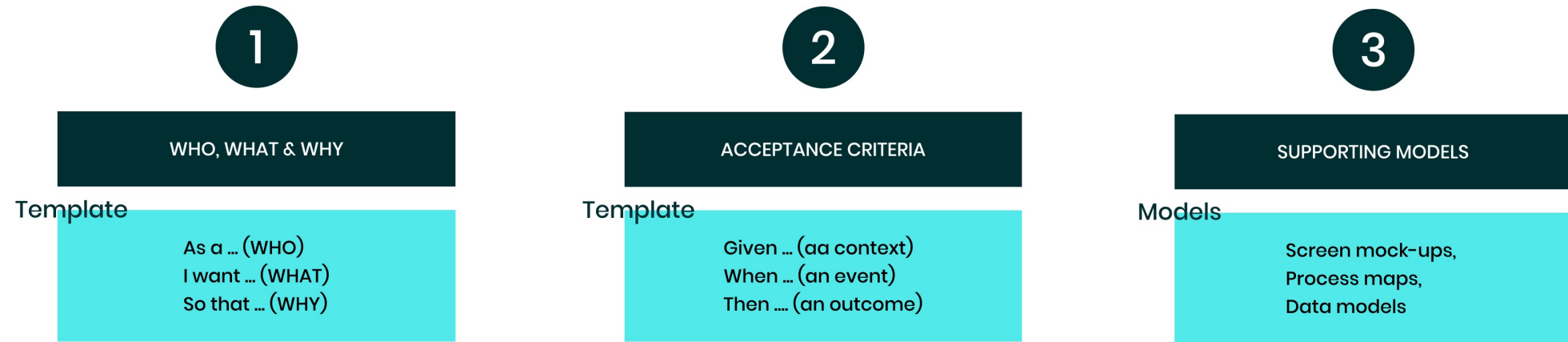


# BUILD GOOD USER STORIES

## *Why good user stories?*

- Delivering a product that the client really needs
- Budget & Time estimation
- Saving time on coding and giving clear tasks to developers
- UI designing





## 3 KEY COMPONENTS OF A GOOD USER STORY

# PRODUCT ENGINEERING WORKSHOP - RECAP

