Management Of Networked IoT Wearables – Very Large Scale Demonstration of Cultural Societal Applications
(Grant Agreement No 732350)

D7.1 Test and Integration Plan

Date: 2017-08-31

Version 1.0

Published by the MONICA Consortium

Dissemination Level: Public

Co-funded by the European Union's Horizon 2020 Framework Programme for Research and Innovation under Grant Agreement No 732350

Document control page

Document file: D7.1 Test and Integration Plan v1.0.docx
Document version: **1.0**
Document owner: ATOS

Work package: WP7 – Components & Cloud Integration
Task: T7.1 – Continuous Integration and Support
Deliverable type: R

Document status: ☒ Approved by the document owner for internal review
☒ Approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Jan Benadik (ATOS)	2017-03-17	Initial version
0.11	Jan Benadik (ATOS)	2017-04-03	Short description of possible tools added
0.12	Jan Benadik (ATOS)	2017-04-11	Some parts of environment description added
0.14	Jan Benadik (ATOS)	2017-06-09	Reworked structure, some details added. minor text corrections
0.15	Jan Benadik (ATOS)	2017-06-19	Test definitions added, test tools added
0.16	Jan Benadik (ATOS)	2017-07-06	LinkSmart modules specifications added SCRAL specifications added OneM2M specifications added
0.17	Jan Benadik (ATOS)	2017-07-07	JMeter description added SoapUI description added
0.18	Jan Badinsky (ATOS)	2017-07-12	Reworked structure
0.20	Robert Najsel (ATOS)	2017-07-12	Software quality added
0.21	Jan Badinsky (ATOS)	2017-07-14	IoT Testing added
0.22	Jan Benadik (ATOS)	2017-07-26	Version for first internal review
0.23	Jan Benadik (ATOS)	2017-08-14	Rewritten into new version of MONICA deliverable template – no content changes
0.39	Jan Badinsky, Jan Benadik (ATOS), Peeter Kool (CNET)	2017-08-31	Reworked based on review recommendations
1.0	Jan Benadik (ATOS)	2017-08-31	Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Summary of comments
Arjen Schoneveld (DEXELS)	2017-08-31	Approved.
Dimitris Katsikas (CERTH)	2017-08-31	Approved. In my opinion in order to have a complete integration plan the reviewer would expect to see: • The Backlog of the iterations in order to prove that we do have a solid plan • A simple Risk Plan (general and related to pilots) with corresponding mitigation actions

Index:

1 Executive Summary

In the Integration part of the MONICA project, parts of the platform which are developed in the technical work packages, shall be integrated into one MONICA cloud platform. In general, the integration serves for the composition and concatenation of different components. In MONICA, parts of the platform which are developed in the technical work packages WP4 – WP6, shall be integrated into one platform. An integration plan describes how the different software modules, subsystems and systems will be integrated into a common prototype platform.

The main output of this deliverable is to define a basis for an integration and testing platform for components used, and application developed in defined work packages.

Main inputs to integration - outputs from definition and development in MONICA work packages:

- WP2 – Demand-side Requirements Engineering for Pilots, Architecture Specification
- WP4 – Acoustics Closed Loop Systems
- WP5 – Security Closed Loops Systems
- WP6 – Situational Awareness & Decision Support

The purpose of this document is to define the scope and approach of integration and testing to be performed in collaboration with related partners of the MONICA consortium, including schedule of intended activities. It identifies integration and test items, the features to be integrated and tested, the integration and testing tasks as well as which role will perform which task.

The execution of the integration and testing plan will validate that the various system components interact and pass data across each other as expected and function together cohesively. It focuses on the interface testing in order to validate that the pieces of the system work together in a seamless way.

Testing will be implemented on different levels:

- Unit testing – Testing at the lowest level using a coverage tool. A unit is a piece of code that does not call any subroutines or functions developed in the project.
- Integration testing – Testing of interfaces between components to ensure that they are compatible.
- System testing – Testing of the entire software system.
- Validation – Evaluation at the end of development to ensure requirements fulfilment.

In line with the AGILE approach to software development, integration should be performed incrementally and continuously. By using Atlassian Jira as Agile Board, Git as Source Code Management System, SonarQube as Quality of Code metrics and Jenkins as the Continuous Integration server, it will be ensured that all developers have access to the latest versions of the code base, the quality of the code is on a desired level and everything is well documented.

2 Introduction

The MONICA project aims to provide large-scale demonstrations of multiple existing and new Internet of Things technologies, demonstrate seamless integration with other (i.e. Smart City) platforms through the MONICA enabling toolbox based on Open Architectures and develop a toolbox of development tools and technology enablers for entrepreneurs and developers in order for them to rapidly develop new IoT applications.

The solution will be deployed and tested by six major cities in Europe during their cultural events in years 2018 – 2019:

- Tivoli/Friday Rock, Copenhagen (DK)
- KappaFutur Festival; The Movida, Torino (IT)
- Hamburger DOM; Port Anniversary, Hamburg (DE)
- Nuits Sonores; La Fête des Lumières, Lyon (FR)
- Rhein in Flammen ; Pützchens Markt, Bonn (DE)
- Cricket matches ; Rugby matches, Leeds (UK)

2.1 Purpose and context of this deliverable

One of the key principles of an agile process is doing everything in small steps but continuously. That is, developing in small iterations, estimating small amounts of work and refactoring in small steps.
Every aspect of the development is continually revisited throughout the lifecycle.

The goal of D7.1 Test and Integration Plan deliverable is to ensure that various tools and components are developed consistently and the final MONICA platform is well integrated. This document defines the basis for the integration of components – applications developed and IoT hardware used in the MONICA project. It is based on the current MONICA infrastructure definition (D2.2 The MONICA IoT Architecture Specification).

An integration and testing plan defined in this deliverable is for testing of the platform and its components developed in the MONICA project. Integration and Testing is explicitly decoupled from the "development of the other components". That is, development of each SW component (Decision Support System APPs, Sound control APPs, Common Operation Picture APPs, visitor APPs, wristband tracking, etc.) follows its own methodology. Many of them can be considered as “black boxes”, that will be integrated into the MONICA system.

In this document, we are presenting an integrated test plan for WP4 – WP6 work packages, because of the development of an integrated software platform where the outcomes of work packages interact with each other. The software quality topic is an integral part of the integration strategy.

2.2 Structure of this deliverable

Chapter 3 discusses the Continuous Integration and Testing topics

Chapter 4 gives an overview of the whole MONICA consortium and the partner activity allocation

Chapter 5 describes the Integration Strategy and Plan

Chapter 6 describes the Test Strategy and Plan

2.3 List of Reference Documents

- D2.1 Scenarios and Use Cases for use of IoT Platforms in Event Management
- D2.2 The MONICA IoT Architecture Specification

- D2.3 Initial Requirement Report
- OneM2M TS-0004-V2.7.1 Service Layer Core Protocol Specification:
  - www.onem2m.org/images/files/deliverables/Release2/TS-0004_Service_Layer_Core_Protocol_V2_7_1.zip
- OneM2M TS-0009-V2.6.1 HTTP Protocol Binding:
  - http://www.onem2m.org/images/files/deliverables/Release2/TS-0009-HTTP_Protocol_Binding-V2_6_1.pdf

2.4 Glossary, Terminology, and Acronyms

Term	Definition
Functional testing	Verify functional requirements implementation, stemming from use cases and business rules
Integration Testing	Verify target system interfacing and collaboration against other interacting systems. Normally conducted under individual system-to-system interaction pairs.
Interface Testing	Verify the correct implementation of each of the MONICA sub-systems against the designated message specifications. MONICA sub-systems provides and consumes functionalities with its interacting systems
System Testing	Verify system functional and non-functional requirements implementation
EC	European Commission
N/A	Not Applicable
SUB	Subscription List
UAT	User Acceptance Test
GE	Generic Enabler
SAF11	Propagation model feed functionality of Situational Awareness System
SA, SAS	Situational Awareness, Situational Awareness System
DSS	Decision Support Service
FIWARE	A generic, open standard platform
NGSI	Next Generation Service Interface
OpenStack	Open source software for creating private and public clouds
CSE	Common Services Entity
NSCL	Network Service Capabilities Layer
WSDL	Web Services Description Language
SOAP	Simple Object Access Protocol
UPnP	Universal Plug and Play
…	… can be extended …

3 Approach, Theory and Methodology

3.1 Continuous Integration

3.1.1 Introduction

When starting a new project, a Continuous Integration should be adopted from the beginning of the project. One of the first steps is to automate the building process, introduce automated testing into the building process, identify the major areas where things go wrong and get automated tests to expose those failures.

Continuous Integration (CI) is a software development practice where members of a team integrate their work frequently; usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible (Fowler, 2006). Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly. CI is a popular practice in agile methodology. It means the team should keep the system always fully integrated. Integration can happen many times a day. Here the point is that CI detects enough bugs to be worth the cost. CI minimizes the time spent on searching bugs and identifies compatibility issues very early.

Enterprises that have implemented agile processes have already seen great improvements both in cost & stability and in the utility of the software. Continuous integration highlights the cultural shifts required to become an agile enterprise. The key to building a culture that supports CI is to make sure that it works the first time.

Continuous Integration takes some effort. If you are committed to AGILE, then it is not a luxury but a necessity. The effort put into it will reflect in the quality of code, responsiveness of the team and in the confidence of a job well done. But don't get caught up with the word 'continuous'. As integration occurs only at certain intervals or when triggered by an event, CI is actually not continuous, but it means it always happens routinely (Quotium, 2014).

Continuous Integration is associated with the concepts of eliminating waste and rapid feedback. Waste is considered as something that doesn't add value to customers. The process flow of the teams is depicted as below (Figure 1). The planning and the architecture teams define the capabilities for the system to be developed. The pre-planning period includes the technical planning, needs analysis, requirements articulation and creation of the architecture framework. Inputs into this pre-planning phase include customer needed capabilities and the output is a vision, roadmap, and architecture. During the pre-planning phase, the planning team defines the scope and deliverables of the project, and the architecture team establishes the vision, architecture, and a product backlog.

The output from the pre-planning phase flows into the first iteration. The architecture team updates the capabilities backlog and prepares materials for the implementation team. These materials may include requirements, capabilities, and user feedback. The architecture team may also work with the implementation team to maintain the architecture as the detailed design evolves and assist developers in aligning the product to the proposed requirements. The Integration and Test team implements the test environment. In subsequent iterations, the finished product from all teams will be tested by the integration and test team to verify the requirements both internally and with the customers (Larri Rosser, 2013).

If any changes are required to the architecture, the architecture team revises the architecture in order to support the upcoming capability development. This may include an architecture revision and a database revision. Here the goal is to provide a modular systems architecture that is resilient to change. All iterations will continue until the release is complete. The release is demonstrated to stakeholders & the planning team for review and acceptance. Requested changes are planned into the next release. At the demonstration of the final release, developers check whether the final software meets all requirements.

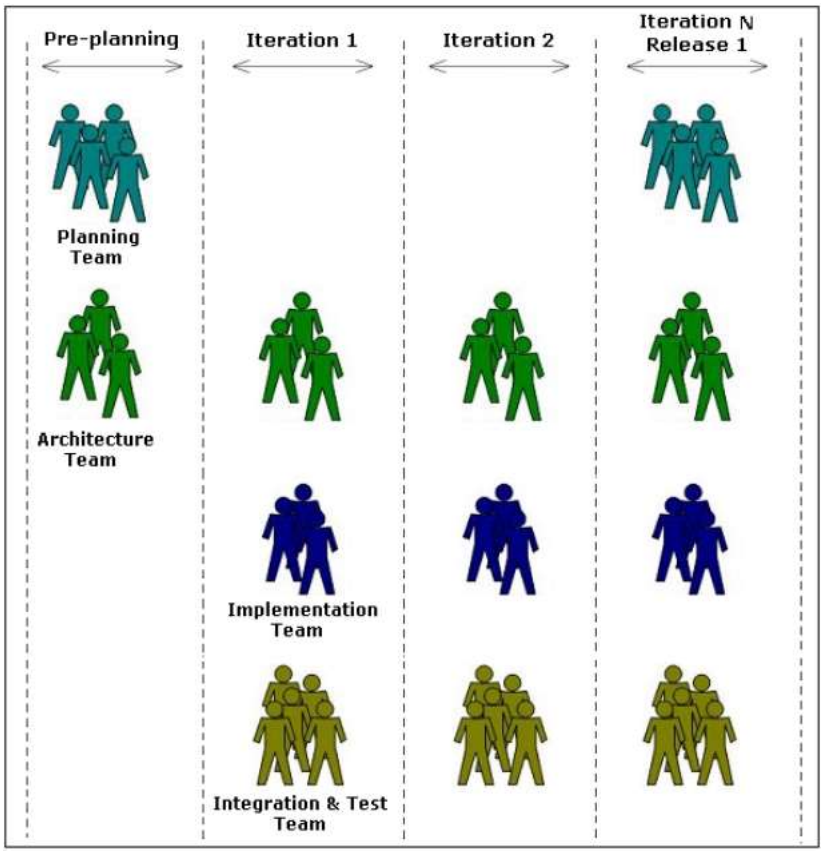

Figure 1: A Scalable Agile process iteration (Quotium, 2014)

3.1.1.1 Benefits of Continuous Integration

Continuous Integrations doesn't get rid of bugs, but it does make them dramatically easier to find and remove (Fowler, 2006). Thus, projects with Continuous Integration tend to have less bugs, both in production and during the development process. On top of CI is Continuous Deployment – aiming at minimizing lead time, the time elapsed between development writing one new line of code and this new code being used by live users in production. Frequent deployment is valuable because it allows your users to get new features more rapidly, to give more rapid feedback on those features, and generally become more collaborative in the development cycle. This helps break down the barriers between customers and development - which are the biggest barriers to successful software development.

3.1.1.2 Building a Feature with Continuous Integration

A source code control system keeps all a project's source code in a repository. The current state of the system is usually referred to as the 'mainline'. At any time, a developer can make a controlled copy of the mainline onto their own machine, this is called 'checking out' [clone in Git]. The copy on the developer's machine is called a 'working copy'. The developer takes his working copy and does whatever she/he should do to complete his task. This will consist of both altering the production code and adding or changing automated tests. Continuous Integration assumes a high degree of tests which are automated into the software.

Once she/he is done (and usually at various points when she/he is working) she/he carries out an automated build on her/his development machine. This takes the source code in his working copy, compiles, and links it into an executable, and runs the automated tests. Only if it everything builds and tests without errors the overall build is considered to be good. Once she/he has made his own build of a properly synchronized working copy she/he can then finally commit (and pushes in case of Git) her/his changes into the mainline, which then updates the repository.

However, this commit doesn't finish his work. At this point we build again, but this time on an integration machine based on the mainline code. Only when this build succeeds we can say that her/his changes are done. There is always a chance that she/he missed something on her/his machine and the repository wasn't properly updated. Only when her/his committed changes build successfully on the integration is her/his job done. Although the developer can manually execute an integration build, it is more valuable to do it automatically by the CI Server.

If the source does not build, the integration build should fail. Either way the error is detected rapidly. At this point the most important task is to fix it, and get the build working properly again. In a Continuous Integration environment, you should never have a failed integration build stay failed for long. A good team should have many correct builds a day. Bad builds do occur from time to time, but should be fixed quickly.

3.1.2 CI Tools

3.1.2.1 Source Control Management technologies

There is lot of Source Control Management technologies in place, many of them distributed under GNU Public Licence. As typical example, Git, Mercurial, and Subversion can be used – for MONICA purpose we use Git.

3.1.2.1.1 Git

Git (GitLab, 2017) is a version control system (VCS) for tracking changes in computer files and coordinating work on those files among multiple people. It is primarily used for software development, but it can be used to keep track of changes in any files. As a distributed revision control system, it is aimed at speed, data integrity, and support for distributed, non-linear workflows. As with most other distributed version control systems, and unlike most client–server systems, every Git directory on every computer is a full-fledged repository with complete history and full version tracking abilities, independent of network access or a central server.

Git is free software distributed under the terms of the GNU General Public License version 2.

3.1.2.2 Source Control Management tools

There is lot of Source Control Management tools in place, many of them freely distributed as Open Source. As an example, SCM-Manager, GitLab, and Atlassian Bitbucket can be used - for MONICA purpose we use SCM-Manager.

3.1.2.2.1 SCM-Manager

SCM-Manager (SCM-Manager, 2017) is a central part of the virtual appliance SCM-Manager Universe. This virtual machine for software development provides a ready to use infrastructure and automated workflows with several features that speed up the development of software and reduce administrative duties. Out of the box support for Git, Mercurial and Subversion.

SCM-Manager is licensed under the BSD-License.

3.1.2.3 Continuous Inspection of Code Quality

Code quality measurement and continuous improvement is not about reactively generating reports and making plans to improve it. Instead of fixing quality issue when they already raise their heads in reports, it’s all about proactively not let them happen at the very first place. Apart from quality plugins used with automated build, IDE plugins and CI plugins help a lot in achieving the holistic agenda of clean code.

The benefits of finding and fixing defects early in the Software Development Lifecycle (SDLC) are widely acknowledged. And these benefits are not limited to quality, but simultaneously have a positive impact on schedule and cost.

There is a lot of different tools and methodologies in identifying and removing defects (Defect Removal Efficiency (DFE)) at the different stages in the SDLC. One of the widely used is SonarQube, which we consider to use for MONICA project development.

3.1.2.3.1 SonarQube

SonarQube (SonarQube, 2017) is an open source platform for continuous inspection of code quality.

It supports languages:
- Java (including Android)
- C/C++, Objective-C, C#
- PHP
- Flex
- Groovy
- JavaScript
- Python
- PL/SQL
- COBOL
- Swift
- etc…

(some of them are commercial).

Offers reports on duplicated code, coding standards, unit tests, code coverage, code complexity, comments, bugs, and security vulnerabilities. Records metrics history and provides evolution graphs.

Provides fully automated analysis / integrates with:
- Maven
- Ant
- Gradle
- MSBuild
- Atlassian Bamboo
- Jenkins / Hudson
- etc...

Is expandable with the use of plugins. Integrates with development environments through the SonarLint plugins:
- Eclipse
- Visual Studio
- IntelliJ IDEA

3.1.2.4 Continuous Integration Software

3.1.2.4.1 Jenkins / Hudson

Jenkins (Vogella, 2017) is an open source automation server written in Java. The project was forked from Hudson after a dispute with Oracle. On February 1, 2011, Oracle said that they intended to continue development of Hudson, and considered Jenkins a fork rather than a rename. Jenkins and Hudson therefore continue as two independent projects, each claiming the other is the fork. As an extensible automation server, Jenkins can be used as a simple CI server or turned into the continuous delivery hub for any project.

Jenkins is one open source tool to perform continuous integration and build automation. The basic functionality of Jenkins is to execute a predefined list of steps. The trigger for this execution can be time or event based. For example, every 20 minutes or after a new commit in a Git repository.

Jenkins monitors the execution of the steps and allows to stop the process if one of the steps fails, it can send out a notification about the build success or failure. Jenkins helps to automate the non-human part of the whole software development process, with now common things like continuous integration, but by further

empowering teams to implement the technical part of a Continuous Delivery. It is a server-based system running in a servlet container such as Apache Tomcat. It supports SCM tools including AccuRev, CVS, Subversion, Git, Mercurial, Perforce, Clearcase and RTC, and can execute Apache Ant, Apache Maven and sbt based projects as well as arbitrary shell scripts and Windows batch commands.

Jenkins functionality can be extended with plugins. Plugins have been released for Jenkins that extend its use to projects written in languages other than Java. Plugins are available for integrating Jenkins with most version control systems and big databases. Many build tools are supported via their plugins. Plugins can also change the way Jenkins looks or add new functionality. There are a set of plugins dedicated for unit testing that generate test reports in various formats (for example JUnit bundled with Jenkins, MSTest, NUnit etc.) and automated testing which supports automated tests. Builds can generate test reports in various formats supported by plugins (JUnit support is currently bundled) and Jenkins can display the reports and generate trends and render them in the GUI.

Released under the MIT License, Jenkins is free software.

3.2 Testing

3.2.1 Introduction

Testing is primarily a risk mitigation function. In short, the role of testing is to advise the program on the quality risks in the delivery, seeking to mitigate the greatest risks as early as possible and to provide information on the residual risks such that a rational judgement between cost, time, and quality of the delivery can be taken.

Testing is defined by the International Software Testing Qualifications Board (ISTQB) as 'The process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.' Testing terminology is defined by ISTQB Standard Glossary of Terms used in Software Testing (see http://istqb.org).

Testing policy:

- Developers should perform Unit tests to verify the software units against the detailed design specifications as standard.
- Separate system test and acceptance test phases should be planned and executed as a minimum. All tests should be planned, repeatable and auditable (i.e. test execution log maintained). Unscripted, but still planned and repeatable, Exploratory Testing is also of value but must be supplementary to the main body of scripted testing.
- Automate testing where appropriate. Aim for a balance between automated and manual tests. Consider automating regression testing where practical return on investment can be achieved, taking account of total contract life, including support.
- Testing for security risk mitigation should be considered during build, initial delivery and regularly during operational use i.e. after infrastructure changes and patch releases.
- Test activities must be monitored to ensure they are proceeding to plan.
- Testing occurs throughout the project lifecycle from inception (reviews) through to production (monitoring). It is the responsibility of the developers, builders, operators, and the test specialists to mitigate quality risks in all their activities.
- Defects found during any testing must be recorded, classified, and actioned appropriately. Root-cause analysis should be undertaken, to both benefit future phases of the project and inform and improve projects in the future.
- Test metrics shall be collected throughout the development and production lifecycle. The metrics collected shall be used to inform subsequent test phases, system releases and projects of the test effort and duration required to achieve a desired quality target measured by residual defects discovered in subsequent tests or operation use.

The Test Strategy is not expected to be updated frequently. It should be defined at a level where it is impervious to changes in plan dates, resources, and detail requirement changes. However, since the Test Strategy is risk based then changes in any content that impacts the risk analysis may require a reassessment of the Test Strategy.

Reviewers of the Test Strategy shall look for adequate coverage of all the topics identified in the template sufficient to ensure efficient and effective guidance for Test Plans and other test collateral.

3.2.1.1 Test Types

- Unit and component test - To verify the software component against the lowest level of documentation defining that component e.g. detailed design specifications. Unit testing focuses on verifying the smallest testable elements of MONICA, namely by testing individual development objects (e.g. individual units of source code) as they are developed during the component build. More specifically, unit testing is performed at the low-level architectural layers.
    - Unit testing is considered part of the development activities and it therefore is assumed that unit testing has been successfully executed by the development team before executing a test campaign on a candidate release.
- Component integration test - Supplier Component Integration Testing verifies that all the components of the supplier solution can interface to and interact correctly with all other components provided by the supplier according to the contractual solution specification documents produced by

the supplier (e.g. functional designs, system specifications). Depending on the nature of the supplier's solution and how completely it fulfils the business requirement, it may be possible to verify some of the business requirements in this test phase. If this can be done it is to be encouraged to detect significant business risks earlier, and fix them with less effort than waiting for later test phases to detect the fault.

- System integration test - System Integration Testing verifies that the delivered systems interface with each other and external organizations/systems as expected. Integration testing aims at confirming that the integration between MONICA Cloud Platform and the other MONICA components external to MONICA Cloud Platform DIGIT (local on-premise deployed) is working as required. This includes the testing of all interfaces including interface logic, interface transmission, and target system processing.
- System test - System Testing verifies that the delivered systems interoperate with each other as expected in the execution of the overall end to end business processes. The testing covers functional and non-functional testing. The risk assessment shall determine the appropriate amount of regression testing to verify that unchanged parts of the system continue to behave as expected.
- User acceptance test - User Acceptance Testing (UAT) is a formal test that proves to the end user that the delivered system is fit for their purpose. The risk assessment shall determine the appropriate amount of regression testing to verify that unchanged parts of the system continue to behave as expected and the existing user expectations can be met. Different types of tests should be performed in the frame of Acceptance Testing:
    - Security testing, to assess data and application configuration protection.
    - Performance testing, to assess time sensitive requirements.
    - Stability testing, to assess workload sensitive requirements.
    - Stress testing, to bring out malfunctions due to lack of (or competition for) resources.
    - Volume testing, to assess the ability to endure workloads during a defined period.
    - Business cycle testing, to ensure the proper functioning of the system over time.
    - Regression Testing, to verify that all functions work properly after code changes.

The responsibility of the testing preparation and execution of these recommended testing type will be specified by the RACI Matrix in 5.3.7.1. The UAT stage will use many of the testing tools and scripts created in the previous stages, particularly the ones used for the system testing and the integration testing, with a special focus on the business processes that are the most common and prone to be used most frequently (not all business processes of system testing and integration testing are tested within a UAT).

3.2.2 Testing Tools

3.2.2.1 Junit

JUnit 5 (Junit, 2017) is composed of several different modules from three different sub-projects:

JUnit Platform + JUnit Jupiter + JUnit Vintage

- **Junit platform** serves as a foundation for launching testing frameworks on the JVM. It also defines the TestEngine API for developing a testing framework that runs on the platform. Furthermore, the platform provides a Console Launcher to launch the platform from the command line and build plugins for Gradle and Maven as well as a JUnit 4 based Runner for running any TestEngine on the platform.
- **JUnit Jupiter** is the combination of the new programming model and extension model for writing tests and extensions in JUnit 5. The Jupiter sub-project provides a TestEngine for running Jupiter based tests on the platform.
- **JUnit Vintage** provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform.

One of the prominent goals of JUnit 5 is to make the interface between JUnit and its programmatic clients – build tools and IDEs – more powerful and stable. The purpose is to decouple the internals of discovering and executing tests from all the filtering and configuration that's necessary from the outside.

JUnit 5 introduces the concept of a Launcher that can be used to discover, filter, and execute tests. Moreover, third party test libraries – like Spock, Cucumber, and FitNesse – can plug into the JUnit Platform's

launching infrastructure by providing a custom TestEngine. A TestEngine facilitates discovery and execution of tests for a particular programming model. For example, JUnit provides a TestEngine that discovers and executes tests written using the JUnit Jupiter programming model.

The ConsoleLauncher is a command-line Java application that lets you launch the JUnit Platform from the console. For example, it can be used to run JUnit Vintage and JUnit Jupiter tests and print test execution results to the console.

3.2.2.2 SoapUI, JMeter, FusionCharts

SoapUI (SoapUI, 2017) is the world's leading Functional Testing tool for SOAP and Web Service testing. With its easy-to-use graphical interface, and enterprise-class features, SoapUI allows to easily and rapidly create and execute automated functional, regression, and load tests. In a single test environment, SoapUI provides complete test coverage - from SOAP and REST-based Web services, to JMS enterprise messaging layers, databases, Rich Internet Applications, automated Functional and Regression Testing. Powerful and innovative features help to validate and improve the quality of services and applications.

The Apache JMeter™ (JMeter, 2017) application is open source software, a 100% pure Java application designed to load test functional behaviour and measure performance. It was originally designed for testing Web Applications but has since expanded to other test functions.

Apache JMeter may be used to test performance both on static and dynamic resources, Web dynamic applications. It can be used to simulate a heavy load on a server, group of servers, network or object to test its strength or to analyse overall performance under different load types.

Apache JMeter features include:

- Ability to load and performance test many different applications/server/protocol types:
  - Web - HTTP, HTTPS (Java, NodeJS, PHP, ASP.NET, …)
  - SOAP / REST Webservices
  - FTP
  - Database via JDBC
  - LDAP
  - Message-oriented middleware (MOM) via JMS
  - Mail - SMTP(S), POP3(S) and IMAP(S)
  - Native commands or shell scripts
  - TCP
  - Java Objects
- Full featured Test IDE that allows fast Test Plan recording (from Browsers or native applications), building and debugging.
- Command-line mode (Non-GUI / headless mode) to load test from any Java compatible OS (Linux, Windows, Mac OSX, …)
- A complete and ready to present dynamic HTML report
- Easy correlation through ability to extract data from most popular response formats, HTML, JSON, XML or any textual format
- Complete portability and 100% Java purity.
- Full multi-threading framework allows concurrent sampling by many threads and simultaneous sampling of different functions by separate thread groups.
- Caching and offline analysis/replaying of test results.
- Highly Extensible core:
  - Pluggable Samplers allow unlimited testing capabilities.
  - Scriptable Samplers (JSR223-compatible languages like Groovy and BeanShell)
  - Several load statistics may be chosen with pluggable timers.
  - Data analysis and visualization plugins allow great extensibility as well as personalization.
  - Functions can be used to provide dynamic input to a test or provide data manipulation.
  - Easy Continuous Integration through 3rd party Open Source libraries for Jenkins

FusionCharts (FusionCharts, 2017) Suite brings events and methods for deep integration with other libraries and web-frameworks. It gives complete control over each step of the charting process, and equips to build the most advanced dashboards for enterprise applications.

3.3 Practices of Continuous Integration and Testing

(Subsections definition – Fowler, 2006)
Maintain a Single Source Repository
Software projects contain many components that need to be co-ordinarily arranged to create a product. Preserving this information when multiple people are involved is a major effort. Therefore, software development teams should develop tools to handle all these issues over the course of the year. These tools - called source code management tools, configuration management, version control systems, repositories, or various other names - are an integral part of most development projects. Costs are not a problem because there are good quality open-source tools available. Once you have a source code management system, make sure that it is a well-known place for everyone. Everything should be in the repository (including test scripts, properties, database schemas, installation/deployment scripts, third-party libraries). The basic rule is that any developer can start a on a project, perform a check-out (clone), and build the system. Only a small number of things should be on virgin machines - usually things that are big, complicated to install and stable. Typically - an operating system, the Java development environment or the database system. IDE configurations are good to place there because people simply share the same IDE settings.

In general - you should store in source control everything you cannot build, but nothing that is a result of the build.

Source Code Quality
There are many ways that static code analysis can help speed up software delivery. It can pick up, as a preliminary to check-in, errors and weaknesses in code that can happen incidentally to even the most experienced developer. It can give the team a measure of „technical debt" (mess in a software code due to implementing new functionality without design redefinition), and remove the obvious "technical noise" (an unnecessary artefacts) from code before it is reviewed.

ISO/IEC 9126 (ISO/IEC 9126) defines a model for software product quality that categorizes software quality attributes into six characteristics:

- Functionality: is the essential purpose of any product or service;
- Reliability: express the ability of the component to maintain a specified level of performance, when used under specified conditions;
- Usability: only exists regarding functionality and refers to the ease of use for a given function;
- Efficiency: is concerned with the system resources used when providing the required functionality;
- Maintainability: the ability to identify and fix a fault within a software
- Portability: refers to how well the software can adopt to changes in its environment or with its requirements.

The quality of the code has major impact on the quality characteristic Maintainability.

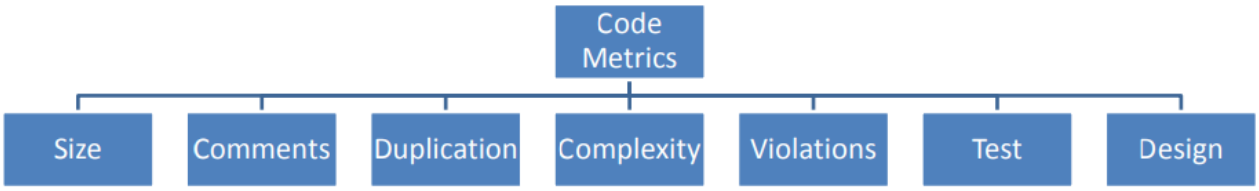

Figure 2: Metrics to measure the quality of the code (focused on Java / C#)

Table 1: Standard targets for Code Metrics

Metric	Target	Motivation
Size	No explicit target. Recommendation is to divide big projects in loosely coupled modules.	Size of code base is derived from other metrics: amount of function points and the complexity of the requirements.
Comments	Density of public documented API: 100% Density of comment lines > 20%	Public APIs define how the code is used in its environment. It is essential information for other programmer to understand the purpose of a class or a method. A certain amount of inline documentation

		helps maintainability. A minimum of 20% is a reasonable target.
Duplication	Density of duplicated lines < 10 %	A target of 10% duplication is reasonable.
Complexity	Nr. of Methods with complexity > 20: 0 Methods with complexity < 11: 80% of all methods (or higher) Average complexity by method < 3	Complexity is a major factor in maintainability. The first two metrics are most important. The average complexity target should be seen as a good indicator.
Violations	Nr. of Major/Blocker/Critical violations: 0 Rules Compliancy > 98%	If there are no major, blocker or critical violations than the target of 98% is easily achieved.
Testing	Coverage of business components: 80%	Test coverage is important for maintainability. A coverage of 80% seems to become an industry standard. Note that the target does not affect UI components
Design	Package cycles: 0	The design of packages should be proper.

Table 2: Recommended process guidelines and tools

Step	Delivery manager	Technical Architect	Software Developer
Project Start		Check customer requirements for code quality, Check for external audit requirements, Define coding standards and priorities (MAJOR, MINOR, etc.), Tailor targets for all metrics	
	Define tailored targets in project plan, Define and plan audits in project plan		
Develop code			Follow guidelines for good code quality
Self-review			Check code for measurements on basic metrics (complexity, duplication, coverage, …)
Code review		Check measurements on dashboard	
Periodic review	Check dashboard weekly		

As for recommended tooling, SonarQube will be used for continuous inspection and evaluation of source code quality. Sonar provides a dashboard for a quick inspection. For a detailed inspection, the source code can be viewed directly in sonar. Measurements are available at the level of project, package, and class. There are also graphical charts available to show the progress over a time period:

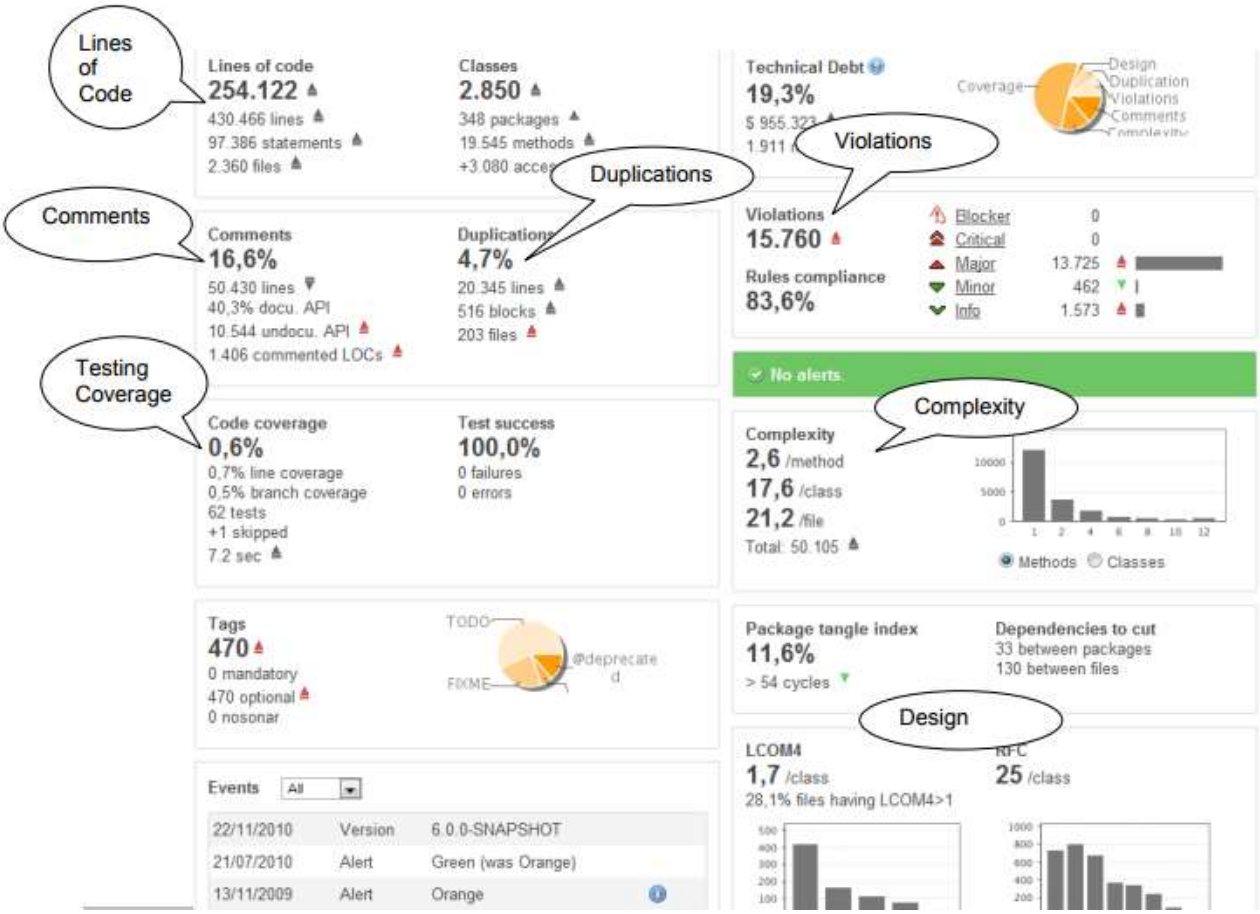

Figure 3: Sonar Dashboard

Automate the Build
The common mistake of automated environments is not to include everything in an automatic assembly. The Build should include obtaining a database schema from a repository and firing it in a production environment - anyone should be able to bring in a virgin computer, check the sources from the repository, enter a single command, and have a running system on it.

Large build often takes time - in the case of minor changes, it is not necessary to do all of these steps. So a good build tool - as part of the process - analyses what should be changed. The common way to do this is to check the dates of the source and object files and compile them only if the source date is later. Dependencies may be complex - if one object file changes, the object that depend on it can also be rebuilt. Compilers can handle these things or they may not. Depending on what you need, different kinds of things may be needed. It is possible to create a system with or without test code or with different series of tests. Some components can be built stand-alone. A build script should allow developers to build alternative targets for different cases (Fowler, 2006).

Make Your Build Self-Testing
A good way to capture errors faster and more efficiently is to include automated tests into the build process. In the case of a self-testing code, a set of automated tests is required to check for a large part of code for bugs. The tests should be possible to start with a simple command and to be self-checking. The result of testing should indicate whether the tests failed, and the failure of the test should cause the build to fail. Of course, it is not possible to rely on tests to reveal everything. But imperfect tests - if runs frequently - are better than perfect tests that do not exist.

Everyone Commits to the Mainline Every Day
Integration allows developers to tell other developers about the changes they have made. Frequent communication allows people to know quickly as changes develop. Integration is primarily about communication.

As with any commit cycle the developer first updates their working copy to match the mainline, resolves any conflicts with the mainline, then builds on their local machine. If the build passes, they are free to commit to the mainline. General rule of thumb is - every developer should commit to the repository every day. In practice, it's often useful if developers commit more frequently than that. The more frequently you commit, the less places you should look for conflict errors, and the more rapidly you fix conflicts. Frequent commits encourage developers to break down their work into small chunks of a few hours each. This helps track progress and provides a sense of progress.

Every Commit Should Build the Mainline on an Integration Machine
Using day-to-day commitments, the team receives often tested builds. In practice, however, the things are getting worse, because of several reasons. One reason is environmental differences between developers' machines. Another is discipline - people do not update and build before they commit. Therefore, you should ensure that integration build is done only on the integration machine and only if the integration is successful, the commitment is confirmed. The developer is responsible for her/his commits, so she/he should monitor the mainline build to fix it if it breaks. As a result, work should not be interrupted until the mainline has been successfully completed with all the commits that have been added in that day.

There are two main ways to ensure this: using a manual build or a continuous integration server.

The manual build approach - it's a similar thing to a local build that a developer does before she/he commits into the repository. The developer checks out the head of the mainline on the integration machine (which now has her/his last commit) and begins integration build. She/he observes its progress and, if the build succeeds, it has made its commit.

A continuous integration approach is automated. The CI server acts as a monitor in the repository. Every time a commit to the repository occurs, the server automatically checks out the sources onto the integration machine, initiates the build, and reports the result of the build to the committer. The commit does not confirm until committer receives a positive notification (usually an email).

Fix Broken Builds Immediately
The whole point of working with CI is that developers are always developing on a known stable base. When the mainline build does break, it's important that it gets fixed fast. Often the fastest way to fix the build is to reverse the latest commit from the mainline, taking the system back to the last-known good build.

Keep the Build Fast
The most important step is to start working on build a pipeline. Because - in fact the sequence of several builds is executed. The commit to the mainline triggers the first build - a commit build. This commit build to the mainline should be done quickly. Therefore, it will take several shortcuts that will reduce the ability to detect bugs. The main topic is to find a balance between the needs to find bug and speed so that a good commit build is stable enough for other people to work on. If the commit build is good, then other people can work on the code with no doubt. Additional machines may run further testing routines on the build taken more time.

In the first phase the team uses a commit build as main CI cycle. The second phase is triggered when possible by picking up the executable file from the latest good commit build for further testing. If this secondary build fails, it means that it does not have sufficient quality, and the team must fix bugs as quickly as possible to keep commit build running. If a secondary build detects a bug, it is a sign that commit build can work with another test. This is an example of a two-stage pipeline, but the basic principle can be extended to any number of stages. These builds (after the commit build) can be done in parallel as well. Using parallel secondary builds you can deploy all further automated tests including performance testing into the standard build process.

Test in a Clone of the Production Environment
The purpose of testing is to prevent/solve - under controlled conditions - any problems that the system may have in production, but there are certain limits. For example, duplicating some production environments can be disproportionately expensive. Despite these limitations, the goal should be to run tests on as much duplicate environments as you can, to understand the risks and accept the differences between the test and the production environment.

Virtualized machines can be saved with all the necessary elements that are embedded in virtualization. Then it is relatively easy to install the latest build and run tests. Additionally, developers can run multiple tests on one machine or simulate multiple computers on a network on the same physical machine. The containers as an alternative approach can be used as well.

Make it Easy for Anyone to Get the Latest Executable
One of the most difficult parts of software development is making sure that developers build the right software. It's very hard to specify what you want in advance and people find it much easier to see something that's not quite right and say how it should be changed.

Everyone can see what's happening
Continuous Integration is all about communication, so you want to ensure that everyone can easily see the state of the system and the changes that have been made to it (Fowler, 2006). One of the most important things to communicate is the state of the mainline build. CI servers' web pages can carry more information than this.

Automate Deployment
To do Continuous Integration multiple environments are needed - one to run commit tests, one or more to run secondary tests. Since executables are moving between these environments multiple times a day, it’s mandatory to do this automatically, so it's important to have scripts to deploy the application into any environment easily.

4 Actors and their integration activities in MONICA Project

Table 3: Actors and their integration activities in MONICA Project

Actor	Integration activities	IoT devices	Integrated modules
FIT	Linksmart deployment support Development of the Geofencing module Pilots deployment support		Linksmart (with CNET) Geofencing module
ACOU	Responsible for running Lyon demonstrations		
ATOS	Components and Cloud integration		
B&K	Sound level meters and GW development and delivery	Microphone sensors	Sound Level Meter GW
BONN	Responsible for running Bonn demonstrations		
CERTH	Situational awareness and decision support module development		Situational awareness and decision support
CNET	Linksmart Deployment support Consumers apps and User Interfaces development Situational awareness and decision support modules (DSS, COP, Guard) development		Linksmart (with FIT) Decision Support System Common Operational Picture Open Air Event App Services MONICA APIs
DEXELS	Testing – the Use Case Definitions Wristbands and Wristbands GE delivery	Wristbands	Wristbands GW
DIGISKY	Delivery of Drones and Airships Install of UWB wireless communication infrastructure	Drones, Airships with cameras and sensors	UWB wireless communication infrastructure
DTU	Sound zone system development and deployment support		Sound Field Control Loop
FHH-SC	Responsible for running Hamburg demonstrations		
HAW	Support for running Hamburg demonstrations Validation and engagement of the Open Data widgets Implementing of the wireless network (Hamburg) Delivery of Sound and Env. Sensors	Sound and environmental sensors	
HWC	Securing of the IoT Network infrastructure Integration of Communication Network and Cyber Security Framework		Cyber Security, Privacy and Trust Framework
IN-JET	Pilots coordination, validation and evaluation		
ISMB	SCRAL deployment support Situational awareness system deployment support		SCRAL Situational Awareness
KK	Support for Copenhagen Pilot		
KU	Security closed loop application development and deployment support		Security Fusion Node
LBU	Support for Leeds Pilot		
MOVE	Support for Torino Pilot		
OPTIN	Delivery of the intelligent eyewear	Intelligent eyewear	

PSG			
RING			
TIVOLI	Responsible for running Copenhagen Pilot		
TIM	Delivery of the OneM2M platform		OneM2M GW and NSCL
TO	Responsible for running Torino Pilot		
VCA	Surveillance applications development	Cameras	Surveillance applications
VH-SJ			
YCCC	Responsible for Leeds Pilot		

5 Integration and Testing

5.1 Assumptions and Constraints

5.1.1 Assumptions

- No native applications for desktop will be developed, only applications with Web UI
- Web UI compatibility of the applications will be tested against the following browsers:
    - Internet Explorer
    - Mozilla Firefox
    - Google Chrome

  on OS's:
    - MS Windows
    - Linux
    - Android
    - IOS
- The native mobile applications will be built for Android as primary platform and IOS as secondary platform afterwards
- The acceptable technologies (C++, C#, html, JavaScript, Python, Java, Docker) and software licenses (GPL, LGPL, MIT, BSD, etc.)
- All documents will be in English
- starting full-length tests in early sprints, even if most of the functionality on both sides of the integration must be mocked out
    - As functionality is added, either sprint-by-sprint or at pre-arranged milestones, the tests can be switched to use the real thing

5.1.2 Constraints

- Weekly team lead meetings are helpful to catch big items, but informal relationships are invaluable for catching semantic differences and focusing on the goal of combined success
- From the early stage, some parts for integration must be simulated

5.1.3 Exclusions

The component development teams and/or component owners are responsible to test and verify the expected functionality of their components, prior any integration testing is conducted

The integration and test plan will not address the following activities:

- Unit testing is under module developer responsibility, excluded from the integration testing
- Completeness and quality of testing data provided
- Hosting hardware, network solution installation, middleware
- Application resources encapsulated by network infrastructure and upon the supporting middleware, and availability of such resources prior any testing session under the target testing environment
- Idle time between two releases or iterations
- Waiting for clarification, inputs
- Change Request received after the completion of the Handover checklist

5.2 Integration and Testing Strategy

Integration is the process of combining software components, hardware components, or both into an overall system. (IEEE Std 610.12, 1990).

The purpose of Integration and Testing Strategy and Plan is to describe how to assemble the product from the product components and ensure that the assembled product operates properly.

Integration Testing is performed to validate that the various system components interact and pass data across each other as expected and function together cohesively. To accomplish this objective, the MONICA

integration team has identified the requirements that are in scope for Integration Testing, which items are out of scope, and the testing activities that will be performed.

The overall testing mission is to verify the correct implementation of the specifications of the MONICA components, as well as, the MONICA platform interconnection and expected information exchange with the systems out of MONICA world (Open Data). One of the testing missions is to guarantee the interoperability aspects of the infrastructure at legal, organizational, semantic, and technical levels. To achieve this, this mission incorporates multiple concerns including:

- find as many bugs as possible
- find important problems, assess perceived quality risks
- advise on perceived project risks
- verify a specification (requirements, design, or claims)
- advise on product quality, satisfy stakeholders
- advise on testing

The integration of the MONICA platform will follow principles described in chapter 3.1 Continuous Integration. Because of the integration happens frequently, the individual extensions and modifications need to be small. The short work cycles require use of an automated methods for source code management, configuration and testing.

Integration, in the case of MONICA, can be considered “development" if integration means integrating all these components to work together. Some components, like DSS really form an “integral” part of the system, these components follow the same methodology as integration. Components that are black boxes, i.e. are/can be used outside of MONICA as well, can follow their own methodology and only their timely availability is important and should follow from the sprint schedule.

Looking at the dynamics of the events that will be using the "MONICA system”, not all events will require all functionality and after the event there will be a lot of feedback. Basically, each event acts as a prototype deployment. The prototype will be evaluated; changes will be made and the next event will use the new prototype. The methodology reflects the dynamics of this discrete operation (during each pilot).

5.2.1 Elements to be integrated

The components to integrate based on an actual state of MONICA architecture definition:

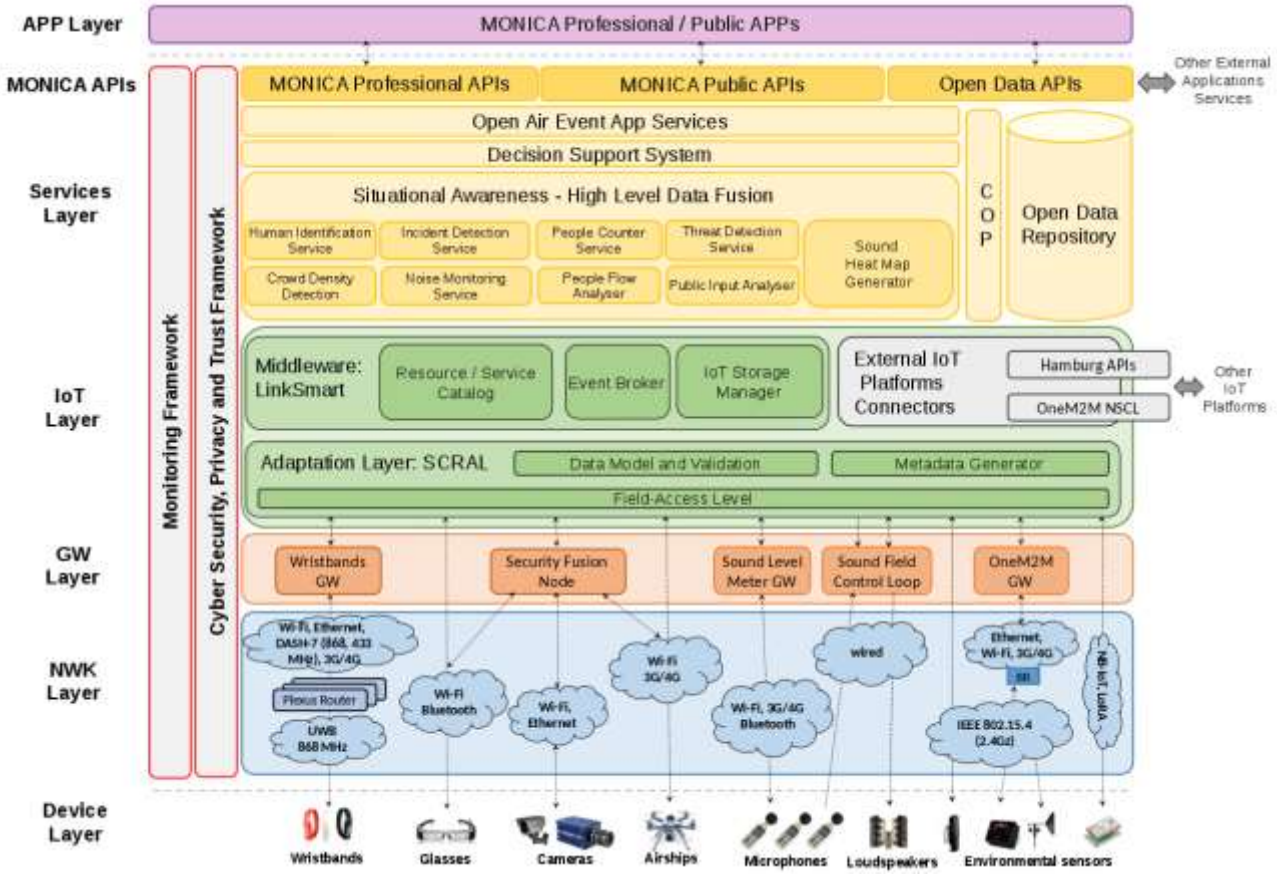

Figure 4: MONICA IoT Architecture

For more details see the document D2.2 The MONICA IoT Architecture Specification.

5.2.2 External and Internal interfaces

List of Interfaces will be updated on the basis of D2.2 The MONICA IoT Architecture Specification.

5.2.3 MONICA Sprints Schedule

The development of the individual MONICA components, or the specific functionalities will be organized into sprints. The Sprints schedule and stories assignment to the specific sprint will be organized by product owner (ISMB). Sprints duration are set to period of one calendar month.

Sprints schedule timeframe falling to specific Releases:

Item	Date / Period
Pilot Demo Release	**M9**
MONICA sprint 1 (M9 DEMO)	25/July - 30/Aug/2017
Annual Platform Deployment 1 Initial release - SCRAL, Linksmart, OneM2M integration	**M10**
MONICA sprint 2	1/Sept – 30/Sept/2017
Prototype release 1 - Crowd & capacity monitoring, Access control, Security incidents	**M12**

MONICA sprint 3	1/Oct – 31/Oct/2017
MONICA sprint 4	1/Nov – 30/Nov/2017
Pilot platform Beta release - Health incidents, Technical incidents, Missing person (Hamburger DOM Beta Release; Rugby Match Beta Release)	**M15**
MONICA sprint 5	1/Dec – 31/Dec/2017
MONICA sprint 6	1/Jan – 31/Jan/2018
MONICA sprint 7	1/Feb – 28/Feb/2018
Pilot platform RC1 release - Sound control loop, Evacuation (Port Anniversary; Rhein in Flammen; Tivoli/Friday's Rock; Nuits Sonores)	**M17**
MONICA sprint 8	1/Mar – 31/Mar/2018
MONICA sprint 9	1/Apr – 30/Apr/2018
Annual Platform Deployment 2	**M18**
MONICA sprint 10	1/May – 31/May/2018
Pilot platform RC2 release (KappaFutur Festival, Hamburger DOM)	**M19**
MONICA sprint 11	1/June – 30/June/2018
Pilot platform RC3 release (Pützchens Markt; Movida)	**M21**
MONICA sprint 12	1/July – 31/July/2018
MONICA sprint 13	1/Aug – 30/Aug/2018
Pilot platform RC4 release (Hamburger DOM)	**M23**
MONICA sprint 14	1/Sept – 30/Sept/2018
MONICA sprint 15	1/Oct– 31/Oct/2018
Prototype release 2 - Event information, Offences, Traffic Pilot platform RC5 release (Fête des Lumières)	**M24**
MONICA sprint 16	1/Nov – 30/Nov/2018
Pilot platform RC6 release (Hamburger DOM; Rugby Match)	**M27**
MONICA sprint 17	1/Dec– 31/Dec/2018
MONICA sprint 18	1/Jan – 31/Jan/2019
MONICA sprint 19	1/Feb – 28/Feb/2019
Pilot platform RC7 release (Port Anniversary; Rhein in Flammen; Tivoli/Friday's Rock; Nuits Sonores)	**M29**
MONICA sprint 20	1/Mar – 31/Mar/2019
MONICA sprint 21	1/Apr – 30/Apr/2019
Annual Platform Deployment 3 Pilot platform Prerelease1 (KappaFutur Festival, Hamburger DOM)	**M30**
MONICA sprint 22	1/May – 31/May/2019
Pilot platform Prerelease2 - Complaints (Pützchens Markt; Movida)	**M33**
MONICA sprint 23	1/June– 30/June/2019
MONICA sprint 24	1/July – 31/July/2019
MONICA sprint 25	1/Aug – 31/Aug/2019
Pilot platform Prerelease3 (Hamburger DOM; Fête des Lumières)	**M34**

MONICA sprint 26	1/Sept – 30/Sept/2019
MONICA sprint 27	1/Oct – 31/Oct/2019
Final Release	**M36**
MONICA sprint 28	1/Nov – 30/Nov/2019

5.3 MONICA Integration and Test Plan

This chapter describes an outline of the release process of separate components for the purpose of an integration and testing of the MONICA prototype platform. The following chapter describes the stages of the integration process.

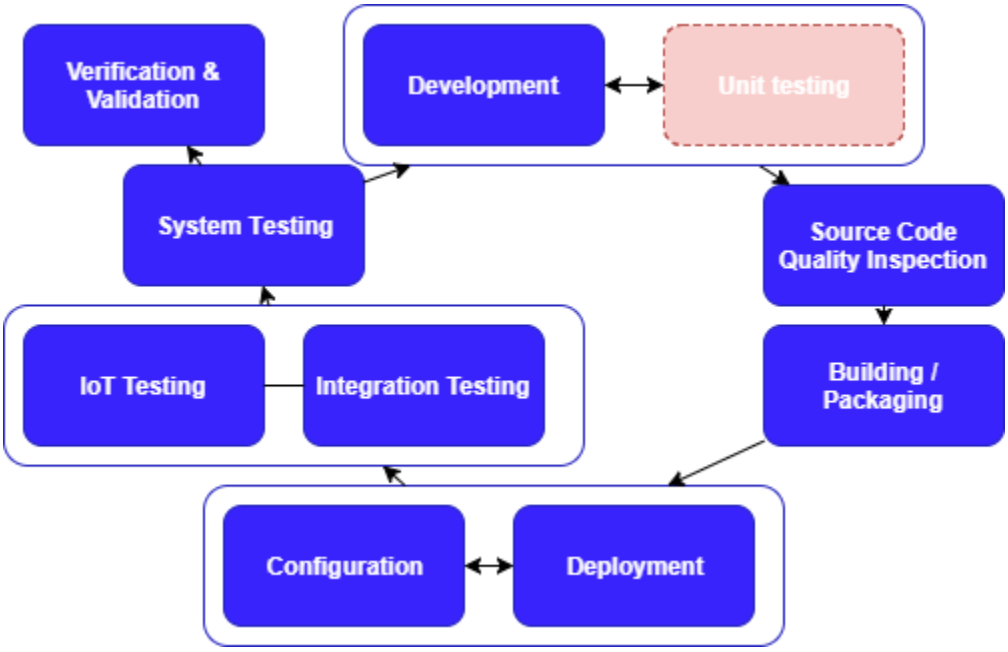

Figure 5: The Integration Process Stages

5.3.1 Integration and Test Phases

A release cycle for the integration process will be based on the basis of sprints schedule, for details see 5.2.3. There is not mandatory to release all components/subcomponents at the end of each release cycle, only components with relevant incremental functionality are mandatory. The main precondition of release is the functionality and usability of components/subcomponents.

At the beginning of the release cycle, every development team participating in the project should commit source code for the specific release to the Central Source Code Repository. The code for the specific release should be stable and unit tested. The Source Code Quality check, Build/Packaging and Deploy/Configuration phases follows, and after these, an Integration and System testing will be realized.

At the end of the release cycle, the validation of product functionality will be held. For the Annual demonstrations (in contrast with the Pilot demonstrations), it will also contain an Approval of Technical manager, and Pilot coordinator at the end of cycle.

5.3.1.1 Development Phase

This phase does not belong to the integration process, in the true sense of word. It is carried out by an individual development teams. But it is necessary to mention this phase, because it is a fundamental precondition for the whole integration process.

Components/subcomponents will be continuously developed on partner's development environments. All partners are obliged to commit their source codes to the Central Source Code (SC) Repository, according to the Sprints schedule, for details see 5.2.3.

- Preconditions: commit to the Central SC Repository after internal Unit tests / Quality checks
- Output: tested source code, prepared for build of a functional and usable component
- Schedule: at least once per release cycle (AD-HOC)

5.3.1.2 Unit Testing

Unit (stand-alone) testing is a precondition, but not the phase of MONICA Integration Testing process. The objective of this stage is to ensure that individual MONICA components / modules are functional, usable and compile able. For integration purposes, the main emphasis in this face should be laid upon interface testing, i.e. compliance of each of the MONICA component against the designated message specifications.
Unit testing will safeguard that the individual MONICA components are well implemented and stable enough to start integration testing.

In this context, integration / testing would like to introduce a streamlined, uniform and consistent way for the Unit Testing stage.

For this purpose, the team will:

- Define a set of common test case specifications that must be successfully executed by each of integrated MONICA component / module. The common test case specifications will focus particularly on the System-Wide Functional Requirements. The definition and execution of test cases for the System Qualities and Security Requirements will remain under the responsibility of Component Developer.
- Facilitate testing by developing and deploying a testing tool that will automate testing as much as possible and will allow integrator / tester to execute the above tests without depending on the actual MONICA services (the testing tool will mock-up it).

Once a Component Owner has implemented and internally tested her / his component, it submits and trigger a unit testing session.

Before and during Unit Testing, the following activities take place:

- Setting up of the target test environment and configuration of the testing tool for each Component Owner (to be performed by Tester).
- Test execution based on the test case specifications and the agreed scope & planning (to be performed by the Component Owner). Each Component Owner will be responsible for logging defects and capturing test results to facilitate the ongoing assessment of their Component. The period dedicated to the execution of the tests will be agreed upfront based on the number of use cases in scope (from … to … working days).
- Note that even though the execution of the Unit testing is under the responsibility of the Component Owner, Tester will provide guidance and be available to support the Component Owner in case of problems
- At the end of the test execution period, Tester completes the test report template and sends it to the Component Owner.
- Once all required tests have been performed, Tester and the concerned Component Owner discuss the test outcomes and make ongoing summary evaluations of the perceived quality of the Component implementation. This debriefing should take place directly at the end of the session or no later than agreed number of working days after the end of the session.

  In a unit test, mock objects can simulate the behaviour of complex, real objects.
  In some circumstances, it may be useful to use a mock object in its place, if:

  - the object supplies non-deterministic results (e.g. the current time or the current temperature)
  - it has states that are difficult to create / reproduce (e.g. a network error)
  - it is slow (e.g. a complete database, which would have to be initialized before the test)
  - it does not yet exist or may change behaviour

(Wikipedia, 2017)

5.3.1.3 Source Code Quality Check Phase

The Source code quality will be inspected at least once per release cycle. The Inspection will be realized by SonarQube, for details see 3.3. A required quality level will be defined (in cooperation with FIT). The Unit test reports of the component from the development phase should be also included. The output of this phase could be a criterion of the Approval phase.

- Preconditions: information about programming languages and their versions per component
- Output: the code quality metrics, list of defects
- Schedule: at least once per release cycle (AD-HOC)

5.3.1.4 Building / Packaging Phase

The next phase after the Quality check is a building / packaging of the components packages. The package of the component should be operational and deployable. Packages should be either built by the partner, or the partners should provide a configuration for the successful build of their components, or partners will configure build the project on the MONICA CI infrastructure. Packages will be stored in the Version Control Repository (VCR - same as the source code repository).

- Preconditions: configuration of build project (if applicable)
- Output: the separate component packages in VCR
- Schedule: at least once per release cycle, after Quality check

5.3.1.5 Deployment Phase

After the collecting of the component packages in the Version Control Repository, components will be deployed to the MONICA environment. After a finishing of the deployment process, a successful deployment of the all components should be checked (the deployment logs check).

In case of the local deployments (demonstration, or components running locally), the provider will be in a way the producer of the component choose. The Cloud deployments will be implemented by the project CI tool (Jenkins).

- Preconditions:
  - Where – pilot environment/Cloud
  - What – all components selected for the Pilot
  - Packages committed to VCR
- Output: Components integrated into single environment
- Schedule:
  - Prototype - at least once per release cycle
  - Pilot – once per year in annual iteration

5.3.1.6 Configuration Phase

The Configuration phase contains the configuration activities of clouds, network (e.g. WIFI, GW), hardware, sensor devices and others physical equipment needed.

- Preconditions: delivered equipment/devices and instructions for configuration
- Output: working and configured environment
- Schedule: same as in Deployment phase

5.3.1.7 Integration Testing Phase

This phase consists of the Integration (Bi-Lateral) testing of MONICA. During this stage, the one module is interconnected with the real second module and the test cases more specifically related to functional testing are executed.

As soon as all the stand-alone unit test cases have been successfully executed, the bi-lateral integration testing stage can begin.

Before and during integration testing, activities like the ones for unit testing are performed (for further details, please refer to the unit testing activities above):

- The setting up and the configuration of the test environment (performed by Tester)
- The test execution (performed by Tester and Component Owner)
- Upon completion, the test report template is completed and delivered to Component Owner
- Debriefing session between Tester and the concerned Component Owner to discuss the test outcomes

The integration tests will be realized after the successful deployment of all components to the specific environment.

- Preconditions: successful deployment of all required components
- Output: tested system prepared for Validation phase
- Schedule: at least once per release cycle, after Deploy

5.3.1.8 IoT Testing

In case of the MONICA project the IoT testing in the distributed environment involving the cloud side and the remote actors, sensors, data streams etc., is one of the crucial points. The IoT testing covers both applications/services and devices as well.

Typical IoT applications should:

- to gather sensor data - a variety of inputs like atmospheric pressure, humidity, noise level, motion tracking, …
- interoperate with different types of hardware, like Arduino boards or Raspberry Pi, others more unusual or context-specific, like video camera, intelligent glasses or microphones
- interoperate with cloud-based servers, web applications or mobiles from which the IoT devices can be monitored and controlled
- communicate via API to enable routine data and device diagnostics pulls to cloud servers, as well as functionality manipulation

Part of the components developed in MONICA project are intended to be used in an IoT scenario meaning that they cannot make assumptions about how they are being used and by which applications. Therefore, they need also to be tested from an IoT perspective and to conform to established IoT standards and principles.

IoT Testing includes the following aspects:

- Discovery Interoperability - discover which “Things” are available on the network and how to communicate with the “Things”. Involves discovery of existence of a “thing” and which IoT services are offered by the thing
    - Device / Service / Resource availability
        - Before they can be integrated and used by an application they need to be discovered by the application, testing verifies that it complies with the discovery mechanisms defined for the specific IoT platform
            - Testing will be done using the Linksmart DAC Browser
    - IoT Service discovery
        - Objects need to be able to communicate/report which IoT Services they support
            - Testing will be done using the Linksmart DAC Browser
- IoT Service Interoperability
    - Web Services
    - REST Services
        - The state and behaviour of the resource can be monitored through simple http GET, POST and PUT commands
            - Testing will be done using standard web browser extension tool (Firefox and Chrome), or command line REST Client
    - UPnP Services
            - Testing will be done using the Linksmart DAC Browser
    - MQTT Test steps
        - Publish / receive test steps
            - This test step publishes / receive a message to / from a topic on the server

        - Drop connection test step
            - Send Disconnect message to MQTT server: send DISCONNECT packet to the MQTT server and then close the network connection (normal behaviour)
            - Close network connection: To simulate some network or client related problems. It is expected that the MQTT server will publish an appropriate Message if it was specified for the connection in that case
- IoT Eventing Interoperability - involves several aspects that need to be tested:
    - Event format
        - Testing will be done by intercepting events and checking them against the agreed schema
    - Event Meta content
        - Testing will be done by message interception and validated by checking the values provided
    - Event payload content
        - Testing will be done by message interception and validated by checking the values provided
    - Event frequency
        - Testing will be done by using the Event Trace and Debug Tool
            - The Event Trace and debug tool can be independently turn on/off side by side with the Linksmart Event Manager and provides the capabilities of eavesdropping on all event communication at a Linksmart Event Manager
                - The Event Trace and Debug Tool is used as a standard event consumer from the Linksmart Event Manager, but it listens to all events without any filtering and stores them in a database
- IoT Exception and Error Handling - using physical tests like simply shutting down or unplug devices from their power and observe the corresponding IoT software behaviour
- IoT Data Security and Privacy Testing - ensure the IoT objects that have received data over a secure channel don´t store or propagate this data in an unsecure manner

IoT Testing will be held concurrently with integration testing:

- Preconditions: successful deployment of all required components
- Output: tested system prepared for Validation phase
- Schedule: at least once per release cycle, after Deploy

5.3.1.9 System Testing Phase

This phase aims to verify the target system End-To-End functionality across the totality of interacting systems realizing MONICA composite system requirements. Testing verifies that the delivered components interoperate with each other as expected in the execution of the overall end to end business processes. The testing covers functional and non-functional testing.

The details of the activities related to End-To-End System testing are currently under development and therefore out of scope of this version of the document.

The system testing phase will proceed after an Integration testing. In this phase the complex functionality of the specific domains/loops, as well as functionality of the whole system will be tested.

- Preconditions: successfully integrated environment with test set of data
- Output: operational integrated system
- Schedule: at least once per release cycle

5.3.1.10 Validation Phase

At the end of a release cycle, the evaluation of the system functionality (if the system complies with the requirement and perform functions for which is intended), will be held.

- Preconditions: successfully integrated environment with test set of data
- Output: operational integrated system
- Schedule: at the end of each release cycle

The validation criteria are stated in the previous phases:

- Unit tests
- Source code quality metrics
- Integration tests
- System tests

According to the milestones 11 and 12 – “MONICA prototype Platform integrated for migration to first and second annual pilot demonstrations” will be considered by Technical Mgr. and Pilot Coordinator (M12, M24) on the basis of the outcomes of the previous phases.

5.3.2 Test stages

The test stages should be defined to avoid joint ownership, minimise test management effort and to have a clear set of objectives. The test stage objectives should be defined clearly enough to avoid the risk of unintentional overlaps or gaps in the test coverage. The names, objectives and sequence of test stages are specific to each project.

The activities for MONICA testing will follow the workflow highlighted below:

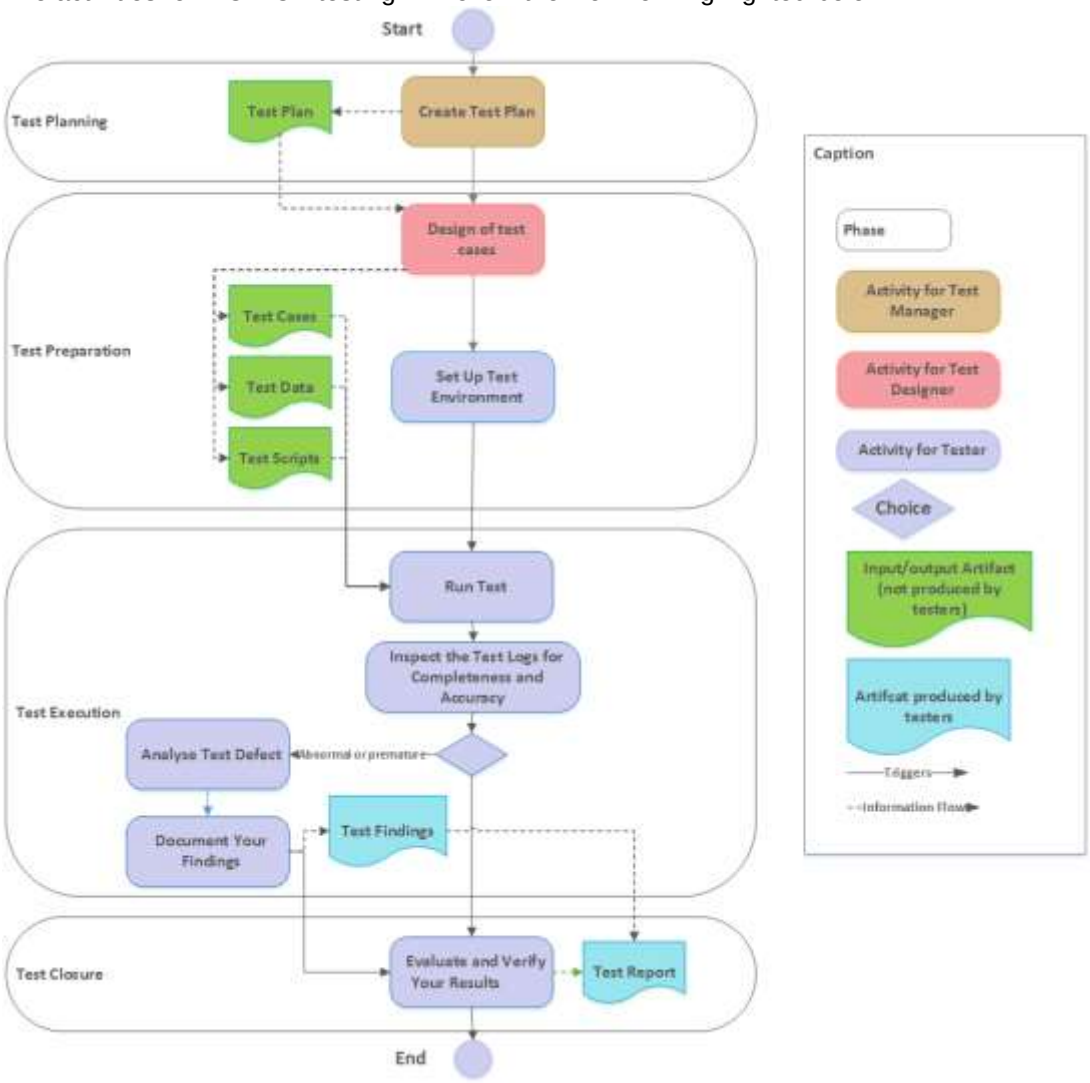

Figure 6: Testing Workflow

The first stage of the testing workflow is Test Planning. The Test Planning stage outlines the overall methodology and strategic direction that the Testing team plans to follow. The tactical details (test identification, testing workflow, etc..) needed for the implementation of the testing strategy and approach are documented in the Test Plan by the test manager.

The second stage is Test Preparation. During this step, the test designers create test cases; prepare test data and scripts while testers set up the test environment.

The third stage is Test Execution. During the testing process, some test cases may fail and defects will be associated with the failures. Defects will be formally tracked and addressed, following the defect management process outlined in section 5.3.6.1.

The fourth stage, Test Closure, is the step which focuses on getting a global overview of the test results, collecting lessons learned and creating Test Reports.

5.3.2.1 Preparation

This includes the following deliverables:

- Integration Test Approach: the current document
- Test case specifications: a document describing the test cases identified for the Unit testing, for the Integration testing and for the System testing
- Test data template: a template to collect the necessary test data to execute the test case specifications
- Test report template: A template to provide an overall status of executed test cases and of identified defects

The preparation activities also include the development of a testing tool will mock-up the behaviour of the components not finished yet. The testing tool will be accessible to the Testers and will allow them to execute, during the unit testing phase, the tests defined in the test case specifications.

The milestone of these preparation activities is the approval by Component Owner of the test approach and scenarios to be executed as well as on the planning of the testing activities. Test preparation activities should be completed prior to Annual Platform Deployment 2017.

5.3.2.2 Execution, Rework, Review, and Retest Procedures

A key success element throughout Integration Testing is the planning, coordination, facilitation, and communication activities that take place among stakeholders on a regular basis, as described in the following sections.

An overview of the planned activities is represented in the Figure below.

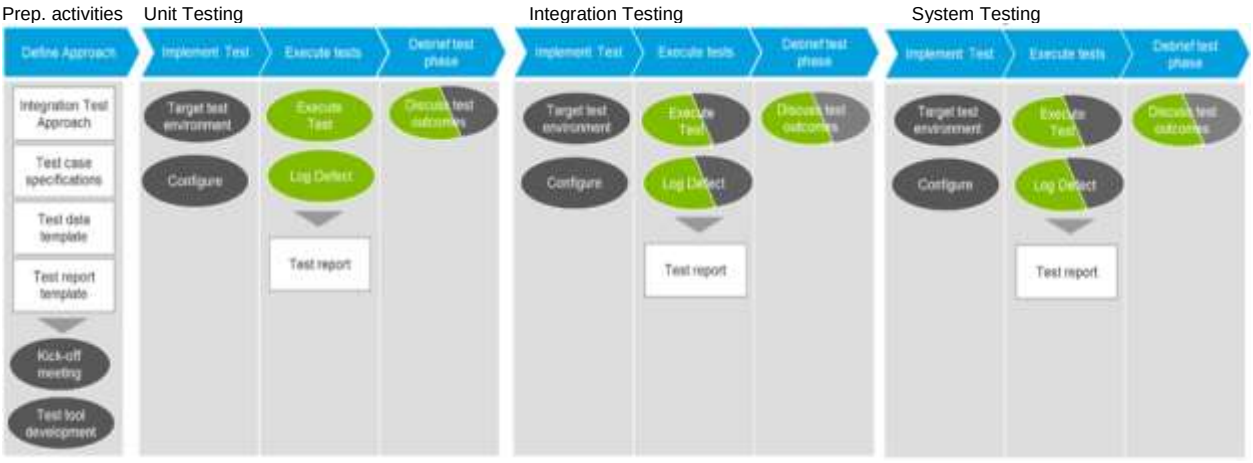

Figure 7: Integration Test Activities

Testing execution phase – test execution will be monitored in JIRA, where all results and bugs / issues founded will be recorded. Bug / issue = a difference between desired and recorded behaviour. Bugs will be reported, analysed, and corrected, the whole process will be fully monitored in JIRA. Bug resolving is related to escalations, priority settings, and meetings planning to risks evaluation (time, severity, affecting of users / other components), extended testing and / or release re-planning.

The test could take the following state:

PASS – Test finished without errors
FAIL – Test was executed, but non-trivial errors occurred
WIP – "Work in progress" – a temporary state of testing scenarios, test not finished yet
BLOCKED – test cannot be finished due to error
NO RUN - Test not started yet
N/A – Test scenario not applicable in current release

Each bug will be classified in JIRA and addressed to resolver. Reports and statistics of testing are the outputs of executing phase.

The severity of issue / bug can be defined as:

- Critical problem (A)
- Major problem (B)
- Minor problem (C)

Detailed definition:
Critical problem (A): functional defects that prevent data processing and end-to-end display, including data security and data corruption, incorrect calculation results, or errors causing the system to reboot, while faulty functionality can not be implemented at the same time Substitute procedures. A critical issue is considered particularly:

- Non-functional SW component as whole, or his major part
- A problem related to major group of users, or
- A problem that is repeatedly feasible or of a permanent nature or the SW component is unusable for the purpose of providing basic user processes and activities for other reasons

Major problem (B): defects that would fall under the Critical Problem (except for issues related to security and data corruption) but to which a workaround can be found to circumvent or prevent a defect. This substitute must be reasonably practicable. A major problem is the failure of the functioning of modules and functions, which severely restricts their use, without restricting the use of the SW component as a whole or its essential parts. The Serious Problem is characterized by a problem that can be repeated or persistent, but the SW component is usable for securing the basic processes and activities of the user with a noticeable impact on the quality of SW component creation or the availability of its functionalities or user comfort.

Minor problem (C): does not meet the Critical Problem or Serious Problem. These are in particular defects that do not prevent the productive use or cause slight discomfort when working with, which is, however, functional.

The tester will determine the severity of the problem based on the principles described above. The severity of the problem (severity) may be reassessed by mutual agreement between the development team and the integrator.

5.3.2.2.1 Integration Test Description

The justified selection of planned and excluded test types out of a standard series of test types is depicted below.

Table 4: Included and Excluded Test Types

Testing Type	Included	Motivation
Architectural Pattern	Yes	Verify system specified architectural pattern realisation

Interface	Yes	Validate system-to-system interactions compliance against the designated interface signatures
Functional	Yes	Verify functional requirements implementation, stemming from use cases and business rules
Security	No	Verify conformance with security requirements and system business function and data access. In particular confidentiality, integrity, availability, authentication, authorization, non-repudiation, time stamping, security logging.
Implementation	Yes	Verify technology specifications implementation and conformance. It is performed during the development
Recovery	No	Ensure software system can be recovered upon failure of hardware, software, or network malfunctioning with undue loss of data or data integrity
User Interface	No	Verify graphical user interface access and navigation across application functionality
Business Cycle	No	Verify system functionality encapsulating time constraints or execute over an extended period. This includes tests for use cases where expiration date or timestamps are involved.
Performance	No	Verify compliance with specified response times, transaction rates, and other time sensitive requirements
Load	No	Verify system stability under nominal workloads
Stress	No	Verify system stability under stress conditions (e.g. workload limits)
Volume	No	Determine the limits of the system subjected to large volume of transactions
Configuration	No	Verify system operation under different hardware and middleware configurations
Installation	Yes	Ensure required hardware, middleware, and application resources are available and operate correctly
Database Integrity	No	Ensure seamless operation of databases and database processes as separate systems At this stage, it is not foreseen to verify database by other means than the requirements concerning data integrity
Regression	Yes	No defects are introduced upon an implementation update or change request realisation

5.3.2.2.1.1 Architecture Patterns Testing

Architecture patterns testing refer to the process of testing the implementation of architectural decisions. The goals of this type of test are to verify that the architectural layers are function as specified and that all messaging flows work properly. This process helps to improve the quality and interoperability of architectural components within the MONICA.

Table 5: Architecture Patterns Testing

Test Objective(s)	Verify that the architectural patterns are correctly implemented
Technique	Execute the list of test cases defined. - This list of test cases covers architectural decisions regarding application architecture identified in D2.2 The MONICA IoT Architecture Specification Exercise each use-case scenario's individual messaging flows and features, using valid and invalid data, to verify that: • The expected results occur when valid data is used in all test cases • The appropriate error or warning messages are generated when invalid data is used • Each business rule is properly applied The appropriate information is retrieved, created, updated, and deleted.
Test Resources	SOAP UI
Exit Criteria	All planned tests have been successfully executed.

Special Considerations	N/A

5.3.2.2.1.2 Interface Testing

Verifies of the Web Service. These tests verify that the Web Service provides the appropriate response to predefined inputs.

Table 6: Interface Testing

Test Objective(s)	• Verify the interface integration between Gateway and Adaptation Layers, as well as Adaptation and Middleware Layers, Middleware, and DSS Layers, DSS and APP Layers (this includes the testing of all interfaces including interface logic and interface transmission) • Reduce the likelihood of integration defects being identified during Acceptance Testing or within the production environment following implementation
Technique	Interface testing involve different kinds of tests. These include but are not limited to: • Baseline Tests: These tests execute each method in isolation, commonly focusing on boundary conditions. Baseline test are performed for synchronous and asynchronous communication. • Data-Oriented Tests: Input and output messages are tested to validate both syntactic and semantic interoperability. Valid data and forced error Tests (SOAP Fault error, empty content, and content exceeding maximum limits) should be executed.
Test Resources	soapUI/loadUI and JMeter can be utilized as the backbone of Interface testing framework
Exit Criteria	Test case is successful.
Special Considerations	N/A

5.3.2.2.1.3 Functional Testing

Focuses on any requirements for test that can be traced directly to use cases or business functions and business rules. The goals of these tests are to verify proper data acceptance, processing, and retrieval, and the appropriate implementation of the business rules. The following table describes an outline of the testing recommended for each MONICA module.

Table 7: Functional Testing

Test Objective(s)	Verify the expected implementation of the user and system requirements. Exercise target-of-test functionality by ensuring proper MONICA functioning, data entry, processing, and retrieval to observe and log target behaviour.
Technique	Execute the list of test cases defined. This list of test cases covers all the identified and documented functionalities. Exercise each use-case scenario's individual use-case flows or functions and features, using valid and invalid data, to verify that: • The expected results occur when valid data is used in all test cases • The appropriate error or warning messages are displayed when invalid data is used • Each business rule is properly applied

	• The appropriate information is retrieved, created, updated, and deleted
Test Resources	• Compatible Web Browsers • Soap UI (automated testing tool) • FusionCharts (java scripts visualisation tool) • Provided mocks
Exit Criteria	• The execution of each test case is successful per expected result • Divergences are logged in a test report. Whenever necessary, the Development Team provide an analysis of the divergence
Special Considerations	Availability of test data and appropriate test environment.

5.3.2.2.1.4 Implementation testing

Implementation testing refers to the process of testing implementations of technology specifications. This process serves the dual purpose of verifying that the technology specification is implementable in practice, and that implementations conform to the specifications. This process helps to improve the quality and interoperability of implementations. Implementation is mostly performed within the unit testing stage, during the development of the system, to verify the proper implementation of specific modules or units.

Table 8: Implementation Testing

Test Objective(s):	Implementation testing ensures that specifications, standards, policies, conventions, and regulations are respected.
Technique:	For each new, updated or configured component to be verified against its technical specification.
Test resources:	Junit
Completion Criteria:	All planned tests have been executed.
Special Considerations:	None.

5.3.2.2.1.5 Installation Testing

Installation testing has two purposes. The first is to ensure that all required software and hardware resources are available for MONICA (e.g. WEB servers, APP servers, databases, etc.). The second purpose is to verify that, once installed, the software and hardware operate correctly. This usually means running several tests that were developed for Function testing.

Table 9: Installation Testing

Test Objective(s)	Install and assess required software and hardware resources.
Technique	Install package software and run installation validation scripts.
Test Resources	All required software and hardware resources.
Exit Criteria	The main pages are properly displayed.
Special Considerations	N/A

5.3.2.2.1.6 Regression Testing

Regression Testing aims at testing a previously tested program following a modification. The purpose is to ensure that defects have not been introduced or uncovered in unchanged areas of the software, because of the changes made. It is performed when the software or its environment is changed.

In the context of MONICA, it is not applicable for the first release of the system. However, the entire set of test cases are performed by default for each version of the System excepted if explicitly mentioned in the release note of the related version.

Table 10: Regression Testing

Test Objective(s)	Verify that all functions work properly after code changes in new builds/releases.
Technique	Run all test cases of the previous build/iteration/release. There is no formal regression testing level (stage) but regression testing is conducted as needed.
Test Resources	Test report.
Exit Criteria	All planned tests have been successfully executed.
Special Considerations	N/A

5.3.3 Ownership of the Integration and Test phases

Clear ownership is essential for an efficient and effective program Integration and Test Strategy especially where the ownership of the integration and test phases is distributed amongst several stakeholders. The integration and test phase owner takes responsibility for producing the test plan, test cases, acquiring the test data, executing the tests, logging defects, retesting and reporting test progress. In the later phases the phase owner may require the support of other stakeholders and may delegate responsibility for the test management to a third party. User Acceptance Testing is a frequent example of a phase that requires participation from many stakeholders. Although the user representative owns the User Acceptance Test, test management may be provided by a third party and the test environments and applications may be supported and maintained by supplier representatives. The Strategy should make the responsibilities of the owner and various stakeholders clear. The names, objectives, entry/exit criteria and ownership of the integration and test phases will be specific to each program. The naming of the integration and test phases should consider the existing naming conventions of the stakeholders to mitigate the risk of confusion and misunderstanding during the implementation of the Strategy.

Some stakeholders may consider regression test as a specific test phase but in the context of the overall program virtually all the test phases should be executing a portion of regression tests. As a general principal, this Integration and Test Strategy Guide considers the regression test as a test technique applied within each test phase rather than a test phase in its own right. The program requirements will determine what regression testing is required and where, when and how that is best executed within each of the test phases. The risk based approach demands that the test phase sequence and content mitigate the greatest risks as early as possible whilst being mindful of practical delivery and resource constraints that may limit the choices for scheduling the test phases. Consider options for improving the robustness of the Integration and Test Strategy to potential changes in dependencies dates and content.

5.3.4 The phase entry and exit criteria

Once testing has begun, the MONICA testing team must track exit, suspension and resumption criteria through the end date identified in the testing schedule. The next criteria are applicable to Unit, Integration and System test.

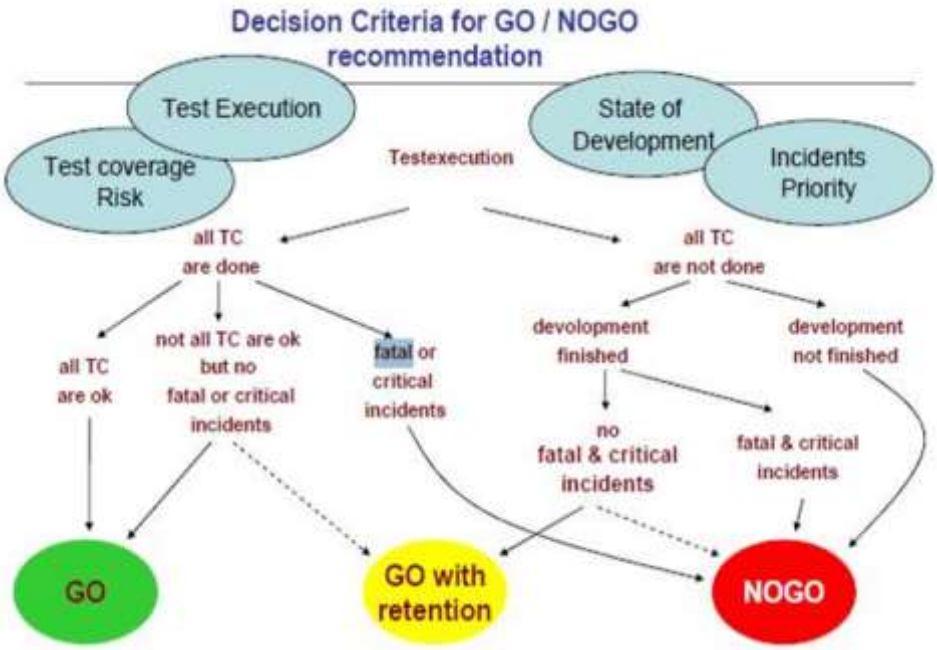

Figure 8: Decision criteria (Source: Atos GDP)

Table 11: Entry and exit criteria

Test phase	Entry criteria	Exit criteria
Unit Tests	• Planning phase has been completed. • System design, technical design and other relevant documents are properly reviewed, analysed and approved. • Business and functional requirements are defined and approved. • Testable codes or units are available. • Availability of test environment. •	• Successful execution of the unit tests. • All the identified bugs have been fixed and closed. • Project code is complete. There are no missing features or elements. •

Component Integration Tests	• Development is done, the code is frozen
	• Integration/System Test environment is set, interfaces are set and functional
	• All testing restrictions are known
	• Documentation for testing is available and final
	• Test plan is accepted
	• Requirements / assumptions of testing are defined
	• Static data and access rights are set
	• Test data are available
	• Unit tests are finished, existing issues are communicated, adressed, the solution is timed
	• Test cases are validated and ready to execute
	• Smoke test has been finished with positive result
		• Integration of new components is verified and functional (limitation of test environment taken into account)
		• The whole content of planned release is implemented (all issues of release are in state „Resolved“)
		• Number of open issues for release:
		  ○ Critical/Showstoppers - 0,
		  ○ Major issue - 0,
		  ○ Minor issue – not exceed 10 issues
		• Version is ready for next level / phase of testing
		• All planned scenarios were executed at least once, the results of testing are available / recorded in JIRA
		• The issues are documented, next steps defined and adressed (including timing)

System Integration Tests	• Pre-Production test environment is set, interfaces are set and functional
	• All testing restrictions are known
	• Static data and access rights are set
	• Test data are available
	• System integration tests are finished, existung issues / bugs are communicated and adressed (exit criteria of previous test phase are fulfiled)
	• Acceptance test cases are ready to execute
		• The version is ready for release
		• Integration is fully functional - all functional requirements were tested / rejected with rational
		• All planned test scenarios were executed at least once, the results of testing are available / recorded in JIRA
		• All issues are documented, next setps defined and adressed (including timing)
		• Number of open issues for release:
		  ○ Critical/Showstoppers - 0,
		  ○ Major issue - 0,
		  ○ Minor issue – not exceed 10 issues

5.3.5 Suspension and Restart Criteria

Suspension criteria must be evaluated during the execution of the test phase. The criteria are based on the following strategy:

- In case the 5 first test cases are not successful; the tester must contact the configuration team to verify the proper configuration of the system
  - If the system is properly configured and the following 5 consecutive tests fail, the tester must contact the test manager to decide if the test campaign must be suspended. This is only applicable for integration and system testing. In case of unit testing, the tester contacts the Master test manager and if they decide to suspend the test campaign, they inform FIT.
  - After 10 test cases marked as failed (non-consecutive), the tester must contact the Master test manager to decide on the possible suspension of the test campaign

After correction of blocking issues or configuration problem, the testing manager gives his approval that the test campaign can be safely resumed. The suspension process is summarized on the picture below:

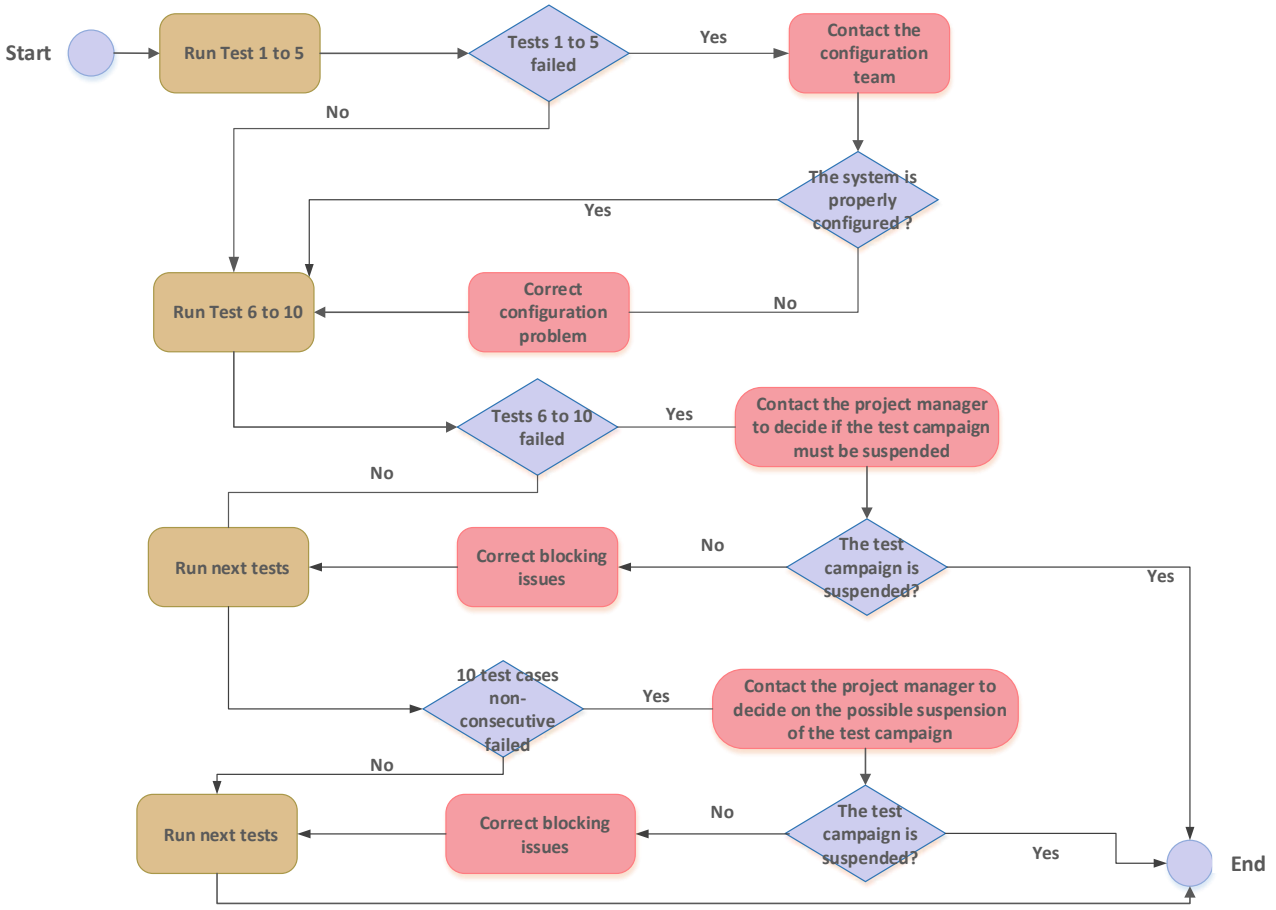

Figure 9: Test Campaign workflow

5.3.6 Products output from the test process

Test work products are divided into inputs and outputs. Input products influencing the Test Strategy should be identified. These may be a variety of requirements, specifications, models, prototypes, corporate standards, industry standards, international standards, legislation, and regulations.

Output products resulting from the processes defined in the Test Strategy consist of a generic set of products and program specific products.

The generic set of potential test products include:

- Test preparation products:

  - Master Test Plan / Detailed Test Plan (if required)
  - Test Conditions
  - Test Cases (including Scripts)
  - Test input data (specifications for production data or files of artificially generated input test data)
- Test execution products:
  - Test Results
  - Incident Reports (an observation of unexpected behaviour not yet classified as a defect, script error, data error, or tester error)
  - Defect Reports
  - Test Metrics
- Test completion products:
  - Test Report
  - Lessons Learned Report (specific to test engagements)
  - Archived testware (for audit record and for handover to production support team).

A RACI matrix can define the roles and responsibilities associated with each test product. Bear in mind that third party suppliers may also be producing test products. In effect, the deliverables from each test phase should be subject to review.

5.3.6.1 Problem Recording and Resolution

5.3.6.1.1 Defect management

Defect management is a lifecycle of the defect. It starts at the time of defect recognition and ends at resolving is closed. From the quality point of view, the defect management consist of:

- Creation of the bug report
- Bug analysis
- Bug assignment
- Bug correction
- Verifying of bug correction

5.3.6.1.2 Creation of the bug report

The process describes identification of the bug and reporting it in JIRA.

Table 12: Creation of the bug report

Input	Identificated bug
Output	Report of identified bug
Responsibility	Developer, Tester

5.3.6.1.3 Bug analysis

Table 13: Bug analysis

Input	Report of identified bug
Output	• Initial version of Report of identified bug • Final version of Report of identified bug • Closed / resolved bug
Responsibility	Authorised person
Related roles	Developer, Tester

5.3.6.1.3.1.1 Detailed specification of Report of identified bug

Activated in case of more details is needed to describe the identified bug.

Table 14: Detailed specification of Report of identified bug

Input	Initial version of Report of identified bug
Output	Detailed Report of identified bug
Responsibility	Developer, Tester
Related roles	Authorised person

5.3.6.1.4 Bug assignment

Assignment of bug correction to authorised person.

Table 15: Bug assignment

Input	Final version of Report of identified bug
Output	Assigment to authorised person to resolve the identified bug
Responsibility	Authorised person

5.3.6.1.5 Bug correction

Root cause analysis and correction time assumption actualise the Report of identified bug.

Table 16: Bug correction

Input	Final version of Report of identified bug
Output	The bug is resolved
Responsibility	Authorised person

5.3.6.1.6 Verifying of bug correction

The author of the Report performs the bug correction verification process. The outcome of this process is either to approve or to deny a bug correction, where the bug is re-assigned to the responsible person with the explanation of the disapproval and added comments to the report.

Table 17: Verifying of bug correction

Input	Final bug resolving
Output	• Approval of bug correction • Rejection of bug correction, modification of Report
Responsibility	Developer, Tester

Table 18: Assessing bugs by severity

Bug classification	Affected area	Bug behaviour
Critical problem (A); Sev1 (Critical)	Testing	• Testing process is fully, or heavy blocked (> 80% test scenarios is blocked)
Major problem (B); Sev2 (Major)	Testing	• Testing is less blocked (< 40% % test scenarios is blocked)
Minor problem (C); Sev3 (Minor)	Testing	• Testing is mostly not affected (< 5% test scenarios is blocked).

5.3.7 Responsibilities

This table shows the staffing assumptions for the test effort.

Table 19: People and Roles

Role	Specific Responsibilities or Comments
Test Manager	Provides management oversight Responsibilities include: • planning and coordination • agree mission • identify motivators • acquire appropriate resources • present management reporting • advocate the interests of test • evaluate effectiveness of test effort
Test Designer	Identifies and defines the specific tests to be conducted Responsibilities include: • identify test ideas • define test details • determine test results • define test automation architecture • verify test techniques • define testability elements • document change requests • evaluate product quality
Tester	Implements and executes the tests Responsibilities include: • implement tests • execute tests • analyse and recover from test failures • document incidents

5.3.7.1 Test RACI matrix

The table below presents the RACI table for the testing types in scope for each testing phase. The following roles in completing tasks are foreseen:

- R - Responsible: person who performs an activity or does the work.
- A - Accountable: person who is ultimately accountable and has Yes/No/Veto
- C - Consulted: person that should provide feedback and contribute to the activity.
- I - Informed: person that should know of the decision or action.

Table 20: RACI Matrix

	R	A	C	I
Unit Test				
Implementation	Component Developer	Component Developer	-	Component Project Manager
System Test				
Business Cycle Testing	MONICA Tester	MONICA Test Manager	MONICA Test Designer	Component Project Manager MONICA Coordinator
Integration Test				
Interface testing	ComponentTester	Component owner	MONICA Test Designer	MONICA Coordinator Component Project Manager
Functional Testing	ComponentTester	Component owner	MONICA Test Designer	MONICA Coordinator Component Project Manager
Acceptance Test				
Performance Testing	MONICA PlatformTester	MONICA Coordinator	MONICA Test Designer	Technical Mgr and Pilot Coordinator
Load/Stability Testing	Tester	Component owner	Test Designer	MONICA Coordinator
Stress Testing	Tester	Component owner	Test Designer	MONICA Coordinator
Volume Testing	Tester	Component owner	Test Designer	MONICA Coordinator
Functional Testing	Tester	Component owner	Test Designer	MONICA Coordinator
Business Cycle Testing	Tester	Component owner	Test Designer	MONICA Coordinator

5.4 MONICA Continuous Integration Environment

For MONICA CI environment, the standard tools from FIT environment were chosen:

- Atlassian JIRA for Issue & Project tracking
  - As an agile board for an application development
  - As an Issue & Project tracking tool for the integration and testing
  - The teams interoperation will be organized in sprints, administered in JIRA
- SCM-Manager for management of Source Code Repository
- SonarQube for Continuous Inspection of Code Quality
- Jenkins Bulid Server
  - Additional plug-ins can extend Jenkins, e.g., for building and testing Android applications or to support the Git version control system.

For more details see 3.1.2.

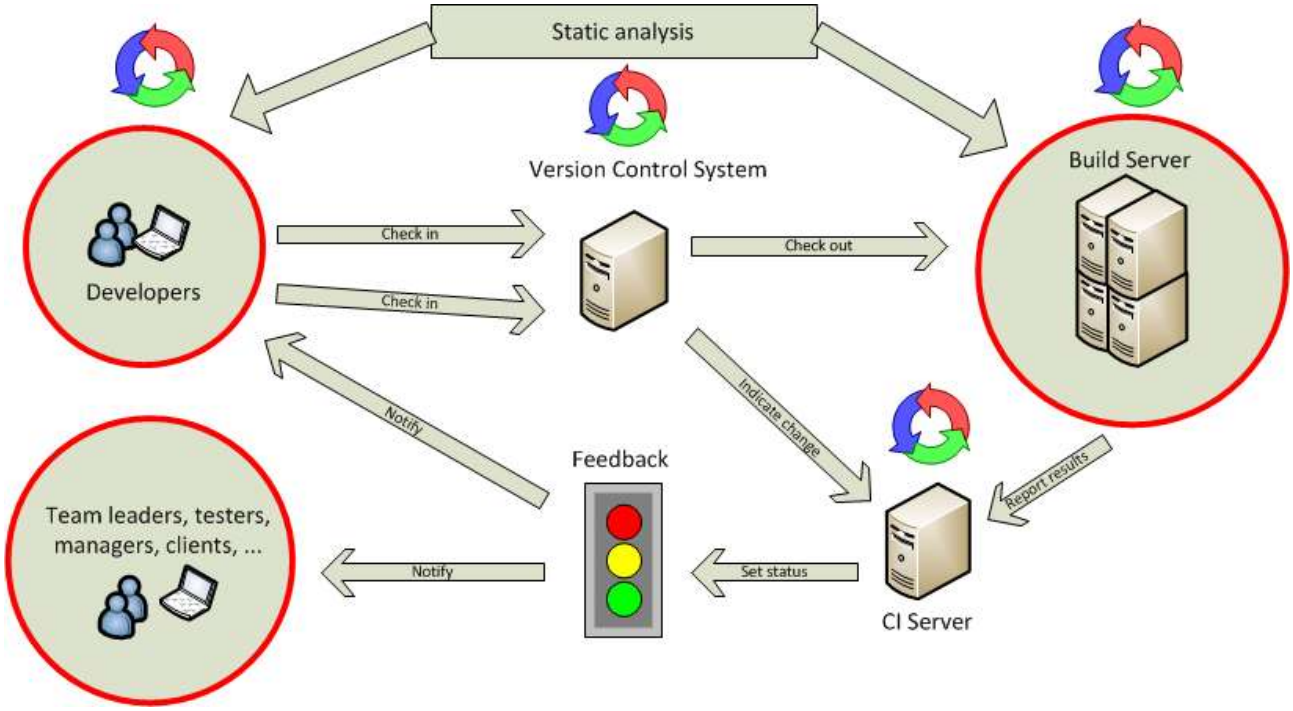

Figure 10: Continuous integration environment

5.5 Test environments

Provision must be made for adequate testing environments. As a minimum, the following separate test environments should be provided:

- Development/Unit Test – this is expected to be placed in partner's environment
- Integration/System Test
- A Pre-Production environment representative of the live environment

Ideally, User Acceptance Testing (UAT) should be undertaken in the Pre-Production environment. Where that is not practical, UAT shall still be executed in a test environment that accurately emulates the user experience using as close to production configuration settings as possible.

Remark: Minimum requirements should not limit mind of thinking in innovative testing solution, e.g. cloud based testing.

5.6 Test data

Once test data are provided, the provided test data set needs to be frozen at the test environment to ensure that it cannot change during the testing session. Tester will use these data to create input and output messages in line with the test cases.

6 Conclusion

This deliverable has described the integration and testing approach, strategy and plan for MONICA project. Integration and Test plan is based on the current definition of the overall architecture; the plan will evolve along with the platform architecture, but the approach and overall plan will remain unchanged. The overall integration strategy has been discussed, and emphasis has been put on the continuous integration model used, which requires good for software code management.

Strategies are influenced by agile software development processes, but the development methodology of components is independent from the integration. The MONICA platform will be a system where applications work with a dynamically changing range of devices and other resources.

Because of the current state of several inputs from the other deliverables (overall MONICA Architecture, details of interfaces definitions still under development), the Platform release plan schedule in 5.2.3 is a draft determined by defined MONICA Milestones, Pilot demonstrations and mapping of the pilot requirements can be changed / redefined based on project evolution, as well as a Risk plan related to pilots, with corresponding mitigation actions, can be part of an individual Prototype deployment preparation.

7 List of Figures and Tables

7.1 Figures

7.2 Tables

8 References

(ISO/IEC 9126) Software Engineering. Product Quality. Part 1: Quality model. Technical Report ISO/IEC 9126-1:2001(E), ISO/IEC.

(IEEE Std 610.12, 1990) IEEE Std 610.12 (1990): IEEE Standard Glossary of Software Engineering Terminology. IEEE Standards Association.

(Fowler, 2006) Martin Fowler, “Continuous Integration”, 01 May 2006

(Quotium, 2014) Continuous Integration in Agile Development, http://www.quotium.com/resources/continuous-integration-agile-development, Accessed 01-04-2017

(Larri Rosser, 2013) Larri Rosser, Phyllis Marbach, Gundars Osvalds, David Lempia, “Systems Engineering for Software Intensive Projects Using Agile Methods”, 2013

(Vogella, 2017) Continuous integration with Jenkins – Tutorial, http://www.vogella.com/tutorials/Jenkins/article.html#using-the-jenkins-build-server, Accessed 2017-04-01

(GitLab, 2017) Git, https://git-scm.com, Accessed 2017-24-04

(SCM-Manager, 2017) SCM-Manager, https://www.scm-manager.org, Accessed 2017-02-02

(SonarQube, 2017) SonarQube, https://www.sonarqube.org, Accessed 2017-02-02

(JUnit, 2017) JUnit. http://www.junit.org, Accessed 2017-06-01

(SoapUI, 2017) SoapUI. https://www.soapui.org, Accessed 2017-07-01

(JMeter, 2017) Apache JMeter, http://jmeter.apache.org, Accessed 2017-07-01

(FusionCharts, 2017) FusionCharts, http://fusioncharts.com. Accessed 2017-07-01

(Wikipedia, 2017) Wikipedia, https://en.wikipedia.org/wiki/Mock_object, Accessed 2017-08-29