# Deliverable
# D5.2 Initial Platform Prototype

| | | |
|---|---|---|
| ***Project Acronym:*** | URBANAGE | |
| ***Project title:*** | Enhanced URBAN planning for AGE-friendly cities through disruptive technologies | |
| ***Grant Agreement No.*** | 101004590 | |
| ***Website:*** | www.URBANAGE.eu | |
| ***Version:*** | 1.0 | |
| ***Date:*** | 31/01/2022 | |
| ***Responsible Partner:*** | ATC | |
| ***Contributing Partners:*** | ENG, TEC | |
| ***Reviewers:*** | Jan Willem (IMEC)<br>Jurgen Silence (AIV) | |
| ***Dissemination Level:*** | Public | x |
| | Confidential – only consortium members and European Commission | |

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101004590

## Revision History

| Revision | Date | Author | Organization | Description |
|---|---|---|---|---|
| **0.1** | 02/09/2021 | Athanasios Dalianis | ATC | ToC |
| **0.2** | 01/10/2021 | Athanasios Dalianis | ATC | Initial input in all chapters |
| **0.3** | 10/11/2021 | Athanasios Dalianis | ATC | Input in chapters 3,4,5 |
| **0.4** | 12/01/2021 | Giuseppe Ciulla | ENG | Input in Section 4 |
| **0.5** | 13/01/2022 | Maritini Kalogerini | ATC | Input in chapters 1,2,6 & addressing comments from TEC |
| **0.6** | 17/01/2022 | Jan Willem | IMEC | Internal Review & comments |
| **0.7** | 19/01/2022 | Jurgen Silence, Andreas Hens | AIV | Internal Review & comments |
| **1.0** | 24/01/2022 | Athanasios Dalianis & Maritini Kalogerini | ATC | Final version |

# Table of Contents

## Table of Figures



## Table of Tables

# List of abbreviations

| Abbreviation | Explanation |
|---|---|
| **API** | Application Programming Interface |
| **CI/CD** | Continuous Integration / Continuous Deployment |
| **CIM** | City Information Model |
| **CSV** | Comma Separated Values |
| **DevOps** | Development Operations |
| **DML** | Data Management Layer |
| **HTTP(S)** | Hypertext Transfer Protocol (Secure) |
| **JSON** | JavaScript Object Notation |
| **REST** | Representational State Transfer |
| **UI** | User Interface |
| **YAML** | YAML Ain't Markup Language |

# 1 Executive Summary

This document summarizes the activities done under Task 5.2. ''DevOps process set up'' up to now, as well as the activities to be performed during the Task's lifetime. More specifically, Deliverable 5.2 "Initial Platform Prototype", defines, gathers, and presents the necessary DevOps processes to be used during the URBANAGE project. For the integration considerations, Agile Software Development Practices are being described and for the Continuous Integration & Deployment Practices a suitable development environment with continuous integration and deployment tools is presented in D5.2. More specifically, an overview of the tools and the CI/CD processes are analysed, with focus on the deployment component, the Hardware and Software requirements, the repositories, the monitoring process, etc. Moreover, this document summarizes an overview of the proposed data integration strategies, posing some preparatory and guiding questions. Finally, an overview of the methodology that will be used, with focus on ISO/IEC 25010:2011, for the system validation of the platform is being presented.

# 2 Introduction

Via URBANAGE activities, the consortium plans to implement a framework for decision making in the field of urban planning, with special focus on facilitating the older people aging well in cities. The process of the decision making is data-driven, taking advantage of massive data production and enhanced analytical capacities in the context of the current digital-era. A decision-support Ecosystem will be the basis of the pre-mentioned framework and it will integrate Big Data Analysis, modelling and simulation techniques with Artificial Intelligence algorithms, adapted visualization methods through Urban Digital Twins and gamification for enhanced engagement purposes. WP5 "Ecosystem & Integration" is the main work package for the provision of the core integration activities, which will address the building of a replicable and extendable core system.

Among other important activities for the creation of the Ecosystem, the DevOps process set up activities aim to describe the necessary tools for Source Code Management, Package Management, Building and Deploying, as well as the pipelines for developing, integrating, and validating the software components. Taking into account the specific requirements of the development methodologies in this project, a 'Rapid Application Development' is proposed to be followed for the integration purposes of the URBANAGE activities, and more specifically the one of the most popular types, the 'Agile Methodology'. In the following sections we summarize all the parameters for setting up the URBANAGE environment as well as the proposed methods for Data Integration and System Validation.

This document is structured as follows:

- Section 3, provides an overview of the URBANAGE development environment in terms of management and implementation methodologies, development and monitoring tools and deployment specifications and processes.
- Section 4, presents briefly the data integration approach for the project
- Section 5, analyzes the system validation methodology that will be followed in URBANAGE
- Section 6, concludes this report.

# 3 Setting up the URBANAGE environment

For the integration purposes of the URBANAGE project we are going to follow the Agile Software Development Practices with frequent integration cycles, rapid prototyping, and close collaboration between self-organizing, cross-functional teams. Based on agile principles, we are also going to apply Continuous Integration techniques for performing automated building, testing and deployment of the provided modules. For adopting Continuous Integration & Continuous Deployment practices we are going to set up a development environment containing a set of continuous integration and deployment tools.

## 3.1 Agile Methodology

Research among the most dominant development methodologies [1] indicates that the most appropriate way of implementing integration mechanisms for the URBANAGE platform would be 'Rapid Application Development'. This implies that a system prototype is implemented, tested, and evaluated in an iterative manner, using short cycles to add functionality to the prototype. This is more suitable for an Innovation action project aiming to deliver a system prototype, since it enables end users to continuously participate in the development of the integration mechanisms and guide the development towards their needs. In this manner, the processes of implementation and definition of the integration mechanisms will proceed in parallel until the end of the project by means of close collaboration between all the teams. One of the most popular types of Rapid Application Development is the 'Agile Methodology', which is associated with a list of terms and rules that must be followed during development as described in the 'Agile Manifesto' [2].

Agile methodology implies and enforces collaboration between self-organizing, cross-functional teams. It promotes adaptive planning, evolutionary development and delivery, a time-boxed iterative approach, and encourages rapid and flexible response to change. Some of the principles of the Agile Manifesto are:

- Welcome changing requirements, even late in development
- Working software is delivered frequently (weeks rather than months)
- Working software is the principal measure of progress
- Customer satisfaction by rapid delivery of useful software
- Close, daily cooperation between businesspeople and developers
- Projects are built around motivated individuals, who should be trusted
- Continuous attention to technical excellence and good design
- Simplicity
- Self-organizing teams
- Regular adaptation to changing circumstances

The methodology workflow could be reflected in the following diagram [3]:

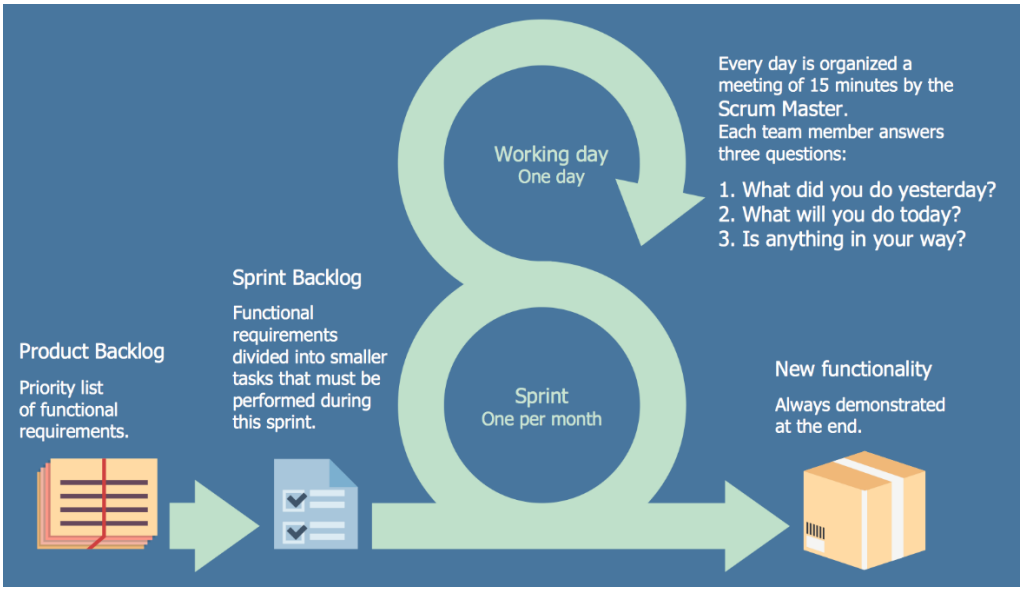


*Figure 1: Agile methodology workflow*

## 3.2 API Guidelines

Most of the components developed in URBANAGE will have APIs exposed. It is important that these APIs follow best practices for better understanding and communication. The minimum API guidelines that need to be followed are summarized in the following list:

- All APIs should use the JSON / REST protocol, over HTTP for internal APIs or HTTPS for publicly available ones
- APIs must return the appropriate HTTP status codes based on status code definitions [4]
- APIs must use the correct service methods [5] for their operations
- APIs must be cacheable when possible
- APIs must support versioning
- APIs must support security measures such as authorization headers
- APIs must support pagination

## 3.3 Overview of the tools and CI/CD process

In this section we present the tools that will be used for the integration of the components of the URBANAGE system. An overview of these tools is presented in the table below.

*Table 1: Integration tools*

| Category | Tool |
|---|---|
| **Task management** | Gitlab |
| **Component packaging** | Docker |
| **Component orchestration** | Docker compose |
| **Component images repository** | Gitlab registry |
| **Cloud services provision** | Hetzner managed server |

| | |
|---|---|
| **Code repository** | Gitlab |
| **Component deployment** | Gitlab pipelines |
| **Code Testing** | JUnit, Mockito, Mocha, Jest, Nose |
| **Monitoring** | Prometheus, Grafana |
| **Code Quality** | SonarQube |

### 3.3.1 Task management

To coordinate the technical efforts between the technical partners and break down the user requirements into technical tasks, in compliance to the agile principles, we have set up an URBANAGE space on Gitlab [6]. Gitlab offers a tool that provides an easy way to create epics, user stories and tasks, to plan agile sprints and assign work to the agile teams, and to keep track of the progress done.

For the purposes of the project:

- Initially the user requirements will be translated to the relevant epics and user stories, and these will be added to Gitlab.
- The technical teams will provide the relevant tasks analyzing the epics and stories.
- These will be refined frequently based on the users' feedback and the projects progress.
- Monthly sprint planning sessions will take place, where each sprint contains the prioritized tasks that can be implemented in the given time, after discussion with all the technical teams.
- Monthly sprint reviews will take place where the results of each sprint are presented to the pilots

### 3.3.2 Deployment

For component isolation and easy deployment, we are going to use Docker [7], whenever possible, that is, all the components developed for the URBANAGE platform, will have to be dockerized.

Docker packages and runs applications in Docker images. A Docker image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Docker images are stored in a Docker registry and in order to run them in virtual or physical machines, the machines need to have installed the Docker Engine software. The running instance of a Docker image is called Docker container. A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another.

For orchestrating and monitoring the Docker containers of the projects we will use Docker compose [8]. Compose is a tool for defining and running multi-container Docker applications. With Compose, we use a YAML file to configure the application's services. Then, with a single command, we create and start all the services defined in the configuration.

The URBANAGE solution will be cloud agnostic, that is, a city at the end of the project will be able to deploy the URBANAGE components, to any cloud provider or cluster of virtual or physical servers, by following the URBANAGE instructions.

#### 3.3.2.1.1 Hardware and Software requirements

The following table summarizes an initial estimation of the component's requirements in terms of hardware, as well as their software dependencies, as these were derived from the component's questionnaires (Annex 1: Technical Questionnaire) and the discussions with the technical team of the project. In the same table we also depict to which pilots a component participates based on the user requirements defined. These requirements were used to define the server specifications for the development environment. More information about these components can be found in D5.1 [9].

*Table 2: Component's hardware and software requirements*

| Component | RAM | CPU | Disk Space | Software dependencies | Pilot |
|---|---|---|---|---|---|
| **Open Data Federator (Idra)** | 8GB | 2 cores | 40GB | Java 8 JDK<br>Apache tomcat 8.5<br>RDF4J Server and Workbench 2.2.1<br>MySQL 5.7.5 | ALL |
| **Orion Context Broker** | 4GB | 2 cores | 20GB | MongoDB | ALL |
| **Connectors for IoT** | 512MB | 1 core | 200MB | Node JS | ALL |
| **Data Repositories Federator** | 16GB | 4 cores | 300MB | Presto DB | ALL |
| **Data Model Mapper** | 4GB | 2 cores | 20GB | Node JS 8.11+ | ALL |
| **CIM** | 8GB | 2 cores | 100GB | PostgreSQL DB, PostGIS<br>GeoServer, Tomcat | Santander |
| **City Visualisation Viewer** | 8GB | 2 cores | 100GB | Java, Javascript<br>Tomcat, PostgreSQL | Santander |
| **Simulation** | 8GB | 4 cores | 2GB | MLFlow, TensorFlow, Databriks | ALL |
| **Optimisation** | 8GB | 4 cores | 2GB | MLFlow, TensorFlow, Databriks | ALL |
| **Predictive / prescriptive analytics** | 8GB | 4 cores | 100GB | TensorFlow, Spark | ALL |
| **Big Data Storage** | 8GB | 4 cores | 100GB | Apache HBase, MongoDB, Kafka, MIN.IO | ALL |
| **Geospatial integration** | 8GB | 2 cores | 2GB | Sedona, Spark | ALL |
| **Timeseries modelling** | 8GB | 2 cores | 1GB | MLFlow, Pandas library, scikits.timeseries | ALL |
| **Workflow management** | 8GB | 2 cores | 20GB | Airflow, PostgreSQL 10+<br>Python 3.6+ | ALL |

| | | | | | |
|---|---|---|---|---|---|
| **Platform UIs & microservices** | 4GB | 2 cores | 2GB | React JS, Spring Boot, PostgreSQL | ALL |
| **Data exploration /visualization** | 2GB | 2 cores | 1GB | React JS, SuperSet | ALL |
| **Identity Manager (KeyCloak)** | 1GB | single core | 1GB | PostgreSQL, Docker | ALL |
| **Message Bus (Kafka)** | 6GB | 2 cores | 10GB | Docker | ALL |
| **Monitoring tools like Prometheus, Grafana, SonarQube** | 2 GB | 2 cores | 2GB | PostgreSQL, Docker | ALL |

For the purposes of the project's platform prototype staging versions, we plan to deploy the URBANAGE components to a managed server provided by Hetzner [10]. More specifically we plan to use the MA-120 server [11] which features 128GB DDR4 RAM, 24-cores AMD EPYC 7401P CPU and 2 x 960GB NVMe SSD disks, which we consider more than enough for the needs of the development environment of the project.

### 3.3.3 Repositories

As a repository for the project's open-source code, a new group at GitLab was created [12].

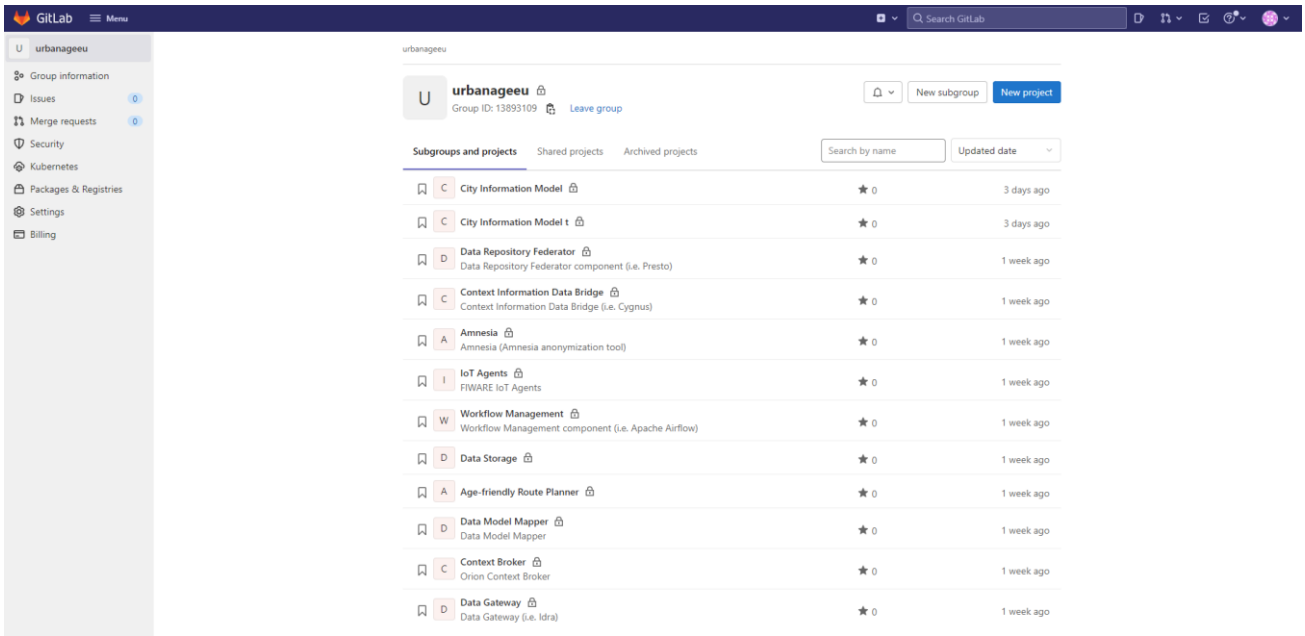


*Figure 2: The URBANAGE GitLab repositories*

GitLab provides a free private space on cloud, with unlimited repositories and users and a set of tools like wiki-based documentation and issue boards. More importantly GitLab offers functionality to automate the entire DevOps life cycle, through its GitLab CI/CD pipelines features.

Technical partners that intend to provide their code under commercial license, are not obliged to use the URBANAGE GitLab repository, however the integration guidelines and quality standards set for all components applies to their own as well.

All components that will be deployed in the URBANAGE platform, must be dockerized. This means that a private Docker registry for the project is required, so that the deployment of the Docker images can be done in a uniform and easy manner.
To this end, we will use the Gitlab container registry [13]. The Gitlab container registry is one of the tools Gitlab provides and is tightly integrated with all the features Gitlab has to offer.

### 3.3.4 Testing and Code Quality

The system validation methodology is described in more detail in Section 5 of the current document. In this subsection we present some of the tools that can be used to retrieve the measurements that will be defined based on the methodology metrics.
All components developed for URBANAGE, will include unit and/or integration tests, to guarantee a higher level of code quality.
Unit tests are automated tests that check if a small part of the application, known as a unit, behaves as it is intended to. In unit testing, any dependencies the unit may have are replaced by "mock" units, that is, units that just return a defined response without implementing any actual functionality.
Integration tests check the behaviour of not only one unit, but a group of units that work together for the completion of a specific functionality. All the dependencies, in this case, including external ones like databases, are real.
Various tools exist for implementing unit and integration tests, in all popular programming languages like:

- Junit [14], Mockito [15] for Java
- Karma [16], Mocha [17], Chai [18] for Node JS
- Unittest [19], nose [20] for Python
- Jest [21], Mocha for React JS etc

Besides the unit and integration tests, the URBANAGE APIs will undergo stress tests to measure their performance under load. A tool that can be used for this purpose is JMeter.
JMeter [22] is designed to load test functional behaviour and measure performance of web applications and web services by defining a set of Web Services Test Plan, which include information like the parameters of the service, the number of concurrent users, the time frame etc.

In order to have a more reliable and globally accepted measure of code quality, for the various quality metrics defined in the validation methodology, the popular SonarQube a quality gateway will be used.
SonarQube [23] is an open-source platform developed for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs, code smells, and security vulnerabilities on more

than 20 programming languages. SonarQube offers reports on duplicated code, coding standards, unit tests, code coverage, code complexity, comments, bugs, and security vulnerabilities.

### 3.3.5 Monitoring

In systems like URBANAGE, it is important to ensure that the different system element services are running smoothly. To this end, the overall performance of the system needs to be monitored continuously and actions to be taken by a system administrator in case of performance degradation or system failure.

The URBANAGE system should be cloud agnostic and since during the project more performance metrics may be defined, additional monitoring tools should be deployed in the platform. Some of these tools can be Prometheus [24] and Grafana [25].

Prometheus is an open-source tool under Apache License, used for event monitoring and alerting. It records real time metrics and stores them in a time series database. It features functionalities like distributed storage, multiple nodes of graphing and dashboarding support and can collaborate with a wide range of tools like Docker, Kubernetes and Grafana.
Grafana is open-source and extendable analytics and interactive visualization web application, that allows a user to query and visualize data, through a set of charts, graphs and alerts, no matter where this data is stored.

### 3.3.6 CI/CD Process

For the purposes of Continuous Integration and Deployment, we are going to use the GitLab pipelines feature. Pipelines are the top-level component of continuous integration, delivery, and deployment and are composed of Jobs that define what needs to be done and Stages that define when the jobs must run.
The Jobs of each Stage can be executed in parallel while the Stages can only be completed sequentially. If all the Jobs of a Stage complete successfully, then the pipeline proceeds to the next Stage. If a Job fails, then the whole pipeline fails.
The pipelines are defined in specific files (gitlab-ci.yml) stored in the root folder of the code repository and involve the creation of integration parameters on the administration pages of GitLab.
The URBANAGE code projects will have to run the pipeline depicted in Figure 3 and has the following Stages:

1. Build the code. This can be considered for example the equivalent of mvn build or npm build in Java and Node JS respectively
2. Run the unit and integration tests defined in the code project. If one of the tests fails, the pipeline fails
3. Produce the quality metrics and push them to the project's SonarQube for further evaluation
4. Create the Docker image of the component and push it to the relevant Docker image registry
5. Deploy the component to the project's server and run docker-compose

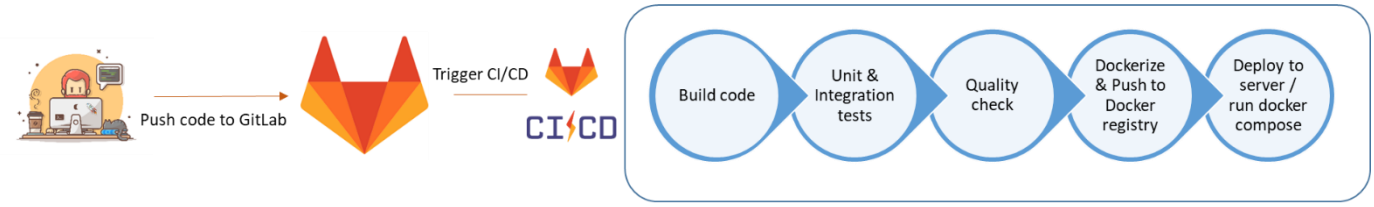


*Figure 3: CI/CD Pipeline*

# 4 Data Integration

Data integration involves combining data coming from different sources to present the data under a unified view.
This is one of the aims of the Data Management Layer (DML) of the URBANAGE platform. This kind of capability is significant when dealing with data from heterogeneous sources, especially when the volume of the data increases (e.g. with big data), and is an essential preparatory step to enable high-level functionalities (e.g. machine learning).
Similar to data aggregation operations [26], also data integrations operations will depend on the specific needs of the use cases and, the analysis to be performed, machine learning algorithms to be trained, etc. So, this kind of operations will be evaluated, planned, and designed case by case.
To this aim, and before the concrete implementation of a data integration process, some preparatory and guiding questions should be answered considering the final results to be achieved (e.g. a data analysis to be performed).
Each question represents a step of a more general process.

- **What data do I need?** To answer this question, it is necessary to have a complete view of the problem to solve and of its context, furthermore, cooperation and discussion between different involved stakeholders is also necessary (e.g. problem owners, domain experts, data analysts, etc.)
- **Is the needed data available?** To answer this question, it is important to identify the data owner that can provide the needed data. Is it the problem owners? Should part of the needed data be provided by other bodies? Is the data public (e.g. Open Data) or released under a specific license?
- **What is the format of the available data?** It is important to ensure that available data is provided with the right format for the specific analysis to be implemented and for the data integration process to be realised; for instance, in many cases, it is necessary to access the data through a structured and machine-readable format (e.g. CSV, JSON, etc.), and the other kinds of formats (like PDF files) imply additional preparatory steps or even the failure of the entire process.
- **How the data is accessible?** It is important to identify the correct procedure to access the data. For instance, if the data is exposed through an API, or a database, or through a static file in a repository, and if any authentication mechanisms are in place.

Once these questions are answered, it is possible to design the more appropriate data integration process for the specific results to be achieved.
To this aim, it is possible to consider different strategies. Some of them are summarised in following table with their main pros and cons.

*Table 3: Data integration strategies overview*

| Strategy | Short description | Pros | Cons |
|---|---|---|---|
| **Manual data integration** | The user manually accesses the data and operate on it all phases of the data integration | The user has the full control over the integration process. | Not possible to operate on huge amount of data. Greater possibility of errors (due to manual operations) |
| **Application-based integration** | The final software applications perform all the operations to realise the needed data integration (locate, retrieve, and integrate data). | All the process is centralised in one application. Reduction of resources needed to perform the data integration process. | Huge complexity of the single software application (e.g. when the number of data sources increases) drive to difficult maintenance. |
| **Middleware data integration** | An application acts as a middleware layer facilitating communication among different systems. | Immediate access to the data offered by the underlying systems. | High technical knowledge for the deployment and maintenance of the middleware. |
| **Uniform access integration** | Data is left on the original source, and it is retrieved and uniformly "displayed" when needed. | There is no need to further store the data. | A high number of data sources can lead to data integrity problems. |
| **Common storage integration (or data warehousing)** | Data is copied and harmonised from the source system into a new one. | Better management of the data (e.g. integrity, uniformity, etc.) and possibility to perform numerous queries and analysis. | Increased storage and maintenance costs. |

DML offers natively the possibility to adopt the *Uniform access integration* and the *Common storage integration* strategies.

Concerning the Uniform access integration strategy, the Data Repository Federator allows to perform distributed SQL queries over multiples federated sources, presenting the results with a uniform representation.

Concerning the Common storage integration strategy, as for the previous one, this is realised thanks to the Data Repository Federator, which results are then harmonised by the Datamodel Mapper and stored into the URBANAGE Data Lake. In addition, context information coming from different sensors and (legacy) IT Platforms are uniformly managed by the Context Broker.

# 5 System Validation

In this section we provide an overview of the methodology that will be used for the system validation of the platform. More specifically we present the ISO/IEC 25010:2011 and explain its quality characteristics. Out of these characteristics we will select the most appropriate ones, in order to form the most suitable quality model for the URBANAGE project and perform our validation tests to the final version of the URBANAGE platform.

## 5.1 Introduction

Software validation is the "confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled" [27]. Since software is usually part of a larger system, the validation of software typically includes evidence that all software requirements have been implemented correctly and completely.

In general, software validation is the process of developing a "level of confidence" that the system meets all requirements, functionalities, and user expectations as set out during the design process. It is a critical tool used to assure the quality of its component and the overall system. It allows for improving/refining the end product.

Software validation is realized through quality models. In the past, different quality models have been proposed, each one of which addresses different quality attributes that allow evaluating the developed software. Some of the most well-known are:

McCall's model of software quality (GE Model, 1977), which incorporates 11 criteria encompassing product operation, product revision and product transition.

Boehm's spiral model (1978) based on a wider range of characteristics, which incorporates 19 criteria. The criteria in both this and the GE model, are not independent as they interact with each other and often cause conflicts.

ISO 9126-1 [28] incorporates six quality goals, each goal having a large number of attributes. These six goals are then further split into sub-characteristics, which represent measurable attributes (custom defined for each software product).

## 5.2 ISO/IEC 25010:2011

Recently, the BS ISO/IEC 25010:2011 [29] standard about system and software quality models has replaced ISO 9126-1. Applying any of the above models is not a straightforward process. There are no automated means for testing software against each of the characteristics defined by each model. For each model, the final

attributes must be matched against measurable metrics and thresholds for evaluating the results must be set. It is then possible to measure the results of the tests performed (either quantitative or qualitative/observed).

The ISO/IEC 25010:2011 standard is the most widespread reference model and includes the common software quality characteristics that are supported by the other models. This standard defines two quality models providing a consistent terminology for specifying, measuring and evaluating system and software product quality, as described below.

### 5.2.1 Quality in use model

The *Quality in use model* is composed of five characteristics that relate to the outcome of interaction with the system and characterizes the impact that the product can have on the stakeholders. It pertains to the notion of *external quality*, i.e. the quality of a (software) product as perceived by its users. External quality assesses the characteristics of the product quality model by *black-box* measurement.

### 5.2.2 Product quality model

The *Product quality model* is composed of eight characteristics that relate to static properties of software and dynamic properties of the computer system. It is intended to measure the *internal quality*, i.e. the quality of the software (and, particularly, its internal components) that eventually **delivers external quality**. Internal quality assesses the characteristics of the product quality model by *glass-box* measurement, i.e. measuring system properties based on knowledge about the internal structure of the software. In each case, the product quality model is adopted. The eight quality characteristics, are further divided into sub-characteristics, as shown in the following figure:

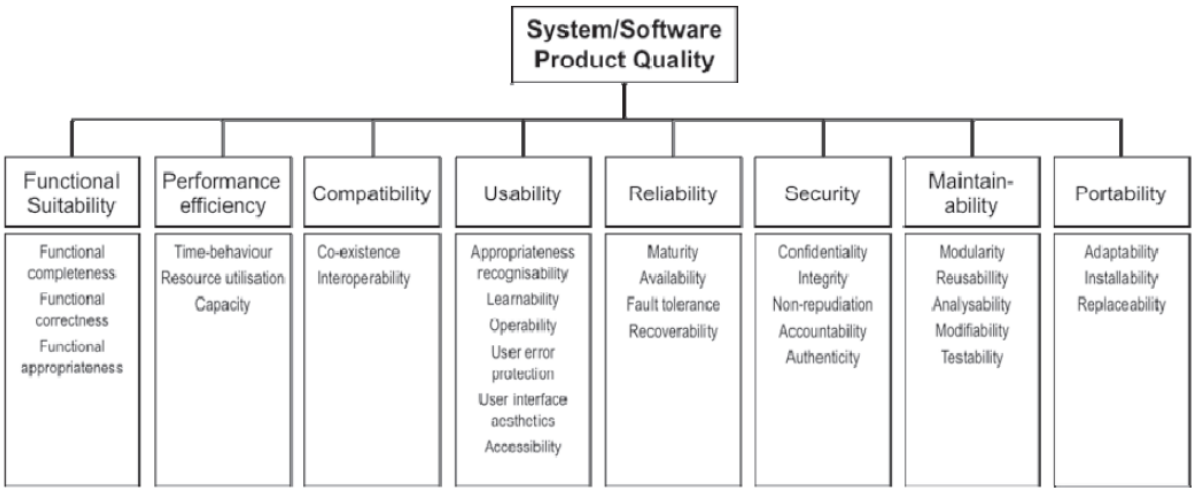


*Figure 4: The ISO/IEC 25010:2011 system/software quality model characteristics*

Although rather generic, not all of the listed quality characteristics might be applicable for our purpose, so a tailor-made subset could be better suited. For each of the sub-characteristics, a metric/measurable attribute will be defined, along with thresholds. These metrics and thresholds are customized for each software product, which in our case is the URBANAGE platform (consisting of individual components). By evaluating these metrics, we will be able to assess the overall quality of our platform and the percent to which we were able to meet the user and technical requirements (reflected to system specifications and functionalities), defined during the design phase of the project.

## 5.3 Designing a quality model

As we have seen, a quality model is the cornerstone of a product quality evaluation system. It determines which quality characteristics will be considered when evaluating the properties of a software product.

### 5.3.1 Understanding the product quality model

The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value. Those stakeholders' needs are precisely what is represented in the quality model, which categorizes the product quality into characteristics and sub-characteristics, as defined below.

### 5.3.2 Functional suitability

This characteristic represents the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions. This characteristic is composed of the following sub characteristics:

- **Functional completeness.** Degree to which the set of functions covers all the specified tasks and user objectives.
- **Functional correctness.** Degree to which a product or system provides the correct results with the needed degree of precision.
- **Functional appropriateness.** Degree to which the functions facilitate the accomplishment of specified tasks and objectives.

### 5.3.3 Performance efficiency

This characteristic represents the performance relative to the number of resources used under stated conditions. This characteristic is composed of the following sub characteristics:

- **Time behavior.** Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements.
- **Resource utilization.** Degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements.

- **Capacity.** Degree to which the maximum limits of a product or system parameter meet requirements.

### 5.3.4 Compatibility

This is the degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment. This characteristic is composed of the following sub characteristics:

- **Co-existence.** Degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product.
- **Interoperability.** Degree to which two or more systems, products or components can exchange information and use the information that has been exchanged.

### 5.3.5 Usability

This characteristic represents the degree to which a product or system can be used by specified users to achieve specific goals with effectiveness, efficiency and satisfaction in a specified context of use. This characteristic is composed of the following sub characteristics:

- **Appropriateness recognizability.** Degree to which users can recognize whether a product or system is appropriate for their needs.
- **Learnability.** Degree to which a product or system can be used by specified users to achieve specific goals of learning to use the product or system with effectiveness, efficiency, freedom from risk, and satisfaction in a specified context of use.
- **Operability.** Degree to which a product or system has attributes that make it easy to operate and control.
- **User error protection.** Degree to which a system protects users against making errors.
- **User interface aesthetics.** Degree to which a user interface enables pleasing and satisfying interaction for the user.
- **Accessibility.** Degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.

### 5.3.6 Security

This is the degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization. This characteristic is composed of the following sub characteristics:

- **Confidentiality.** Degree to which a product or system ensures that data are accessible only to those authorized to have access.
- **Integrity.** Degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data.

- **Non-repudiation.** Degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later.
- **Accountability.** Degree to which the actions of an entity can be traced uniquely to the entity.
- **Authenticity.** Degree to which the identity of a subject or resource can be proved to be the one claimed.

### 5.3.7 Maintainability

This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve, correct, or adapt it to changes in environment, and in requirements. This characteristic is composed of the following sub characteristics:

- **Modularity.** Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
- **Reusability.** Degree to which an asset can be used in more than one system, or in building other assets.
- **Analyzability.** Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
- **Modifiability.** Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
- **Testability.** Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

### 5.3.8 Reliability

This is the degree to which a system, product or component performs specific functions under specified conditions for a certain period. This characteristic is composed of the following sub characteristics:

- **Maturity.** Degree to which a system, product or component meets needs for reliability under normal operation.
- **Availability.** Degree to which a system, product or component is operational and accessible when required for use.
- **Fault tolerance.** Degree to which a system, product or component operates as intended despite the presence of hardware or software faults.
- **Recoverability.** Degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system.

### 5.3.9 Portability

Portability is the degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another. This characteristic is composed of the following sub characteristics:

- **Adaptability.** Degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments.
- **Installability.** Degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment.
- **Replaceability.** Degree to which a product can replace another specified software product for the same purpose in the same environment.

# 6 Conclusion

In this document we summarized the research activities for setting up the working environment for the URBANAGE platform integration. More specifically, Deliverable 5.2 “Initial Platform Prototype”, defines, gathers and presents the necessary DevOps processes to be used during the URBANAGE project. The main findings/recommendations include the adoption of Agile Software Development Practices as well as the adoption of 7 specific API guidelines. Moreover, tools such as Gitlab (Gitlab registry, Gitlab pipeline), Docker, & Docker compose, JUnit, SonarQube etc have been selected for the integration of the components of the URBANAGE system. Also, special focus has been given to the initial estimation of the component’s requirements in terms of hardware and software, as described in Table 2. In addition, the design of a more appropriate data integration process, has been facilitated through the creation of preparatory and guiding questions. Finally, the methodology that will be used for the system validation has been developed under ISO/IEC 25010:2011 standard. In that scope, the design of initial platform prototype and its characteristics and guidelines, have been gathered, analyzed and proposed to the URBANAGE consortium.

# 7 References


[1] http://en.wikipedia.org/wiki/Software_development_methodology
[2] http://agilemanifesto.org/
[3] https://www.conceptdraw.com/How-To-Guide/scrum-workflow
[4] https://restfulapi.net/http-status-codes/
[5] https://restfulapi.net/http-methods
[6] https://about.GitLab.com/solutions/agile-delivery/
[7] https://www.docker.com/
[8] https://docs.docker.com/compose/
[9] D5.1 System Architecture & Implementation Plan
[10] https://www.hetzner.com/
[11] https://www.hetzner.com/managed-server/ma120
[12] https://gitlab.com/urbanageeu
[13] https://docs.gitlab.com/ee/user/packages/container_registry/
[14] https://junit.org/junit5/
[15] https://site.mockito.org/
[16] https://karma-runner.github.io/latest/index.html
[17] https://mochajs.org/
[18] https://www.chaijs.com/
[19] https://docs.python.org/3/library/unittest.html
[20] https://pypi.org/project/nose/
[21] https://jestjs.io/
[22] https://jmeter.apache.org/
[23] https://www.sonarqube.org/
[24] https://prometheus.io/
[25] https://grafana.com/
[26] D3.1 Data Manager Layer. Initial
[27] https://www.complianceonline.com/resources/software-verification-and-validation-requirements-for-medical-device-and-implementation-strategies.html#:~:text=Software%20validation%20is%20the%20%22confirmation,software%20can%20be%20consistently%20fulfilled.%22
[28] https://en.wikipedia.org/wiki/ISO/IEC_9126
[29] https://www.iso.org/standard/35733.html

# 8 Annex 1: Technical Questionnaire

The following questionnaire was distributed to the technical partners of the platform in order to gather information about the components and tools that will be used to compose the URBANAGE platform.

## URBANAGE Technical Questionnaire

**INTRODUCTION**

The following questions are requested to be filled by URBANAGE's technology partners who are going to develop the components in RTD WPs and that will be integrated in the URBANAGE platform. Please provide your answers and you are kindly requested not to respond with a simple yes or no. Try to elaborate on your answers by giving an example of use, to the extent that this is possible.

**QUESTIONS**

1. Questions about your component

    1. Please name your component and provide a short description of functionality you plan to deliver in the project and any foreseen interrelation with another URBANAGE software component.

Component Name:
Component Description:

    2. Do you have an existing background tool that will be adopted in the project to support your component? If yes, please provide the following:

        - Any past project that was used:
        - The component architecture and technologies used in the development:

    3. Did/Will you implement your component from scratch or is it an extension of a third-party platform? If you have extended a third-party platform, please provide at least the following:

        - Name:
        - Version:

    4. Does your component support / require user interaction? If yes, please provide a short description of the expected user feedback.

    5. Have you ever performed a stress test on your component under heavy load? If yes, please provide the test results.

6. Which person in your team is the designated component "owner" (for communication purposes)?

7. If known, please state the names of people actively involved in the development of the component (for communication purposes).
8. Will the component be provided as a service, binary or source code?

9. What will be the license of the delivered component?

10. Are there any restrictions in its use?

11. Will it be made available under an open-source license?

2. Development Needs

1. In which programming languages does your component depend (i.e., java, C++, etc.)?

Please provide at least the following:

- Name:
- Version:

2. Does your component depend on existing design tools (i.e., eclipse, process modelling tool, etc.)? Please provide at least the following:

- Name:
- Version:

3. Which are the hardware requirements for your component to run (i.e., any restrictions in memory use, required CPU, etc.)?

4. Which are the software requirements for your component to run (i.e., operating system, tomcat, apache, etc.)

5. Does your component have any third-party dependencies or use third party libraries? If yes, please provide at least the following:

- Name:
- Version:

6. Can you support automatic builds? Can you support Ant or Maven?

7. Does your component have any database requirements? If yes, please provide the following:

   - required design (i.e., SQL-like, noSQL):
   - database schema (i.e. ER diagram, JSON format):

8. Is there a requirement on a specific Version Control System?

9. Are you going to use a specific tool for the technical verification and evaluation of your component? If yes, which is the tool?

3. Interoperability Requirements

   1. Does your component offer an API to work as a service?

      - **If yes**, please provide the full API Documentation for both input and output streams:
      - **If no**, please describe the communication schema supported by your component:

   2. Can your component support asynchronous messaging?

   3. Does your component need any specific data transformation schema?

   4. Which are the foreseen dependencies with other URBANAGE software components, if any? Please provide at least the following:

      - List of components:
      - Input and / or output requirements:
      - Type and aim of the information that need to be exchanged:

4. Security principles

   1. Which are the security principles (i.e. Principle of Least privilege, establish secure defaults, etc.) that should be taken into consideration to protect your component's data?