DELIVERABLE

D2.2 Initial end-user requirements

Project Acronym:	LAZARUS	
Project title:	pLatform for Analysis of Resilient and secUre Software	
Grant Agreement No.	101070303	
Website:	https://lazarus-he.eu/	
Contact:	info@lazarus-he.eu	
Version:	1.0	
Date:	30/04/2023	
Responsible Partner:	MOT	
Contributing Partners:	ARC, UCM, UNIPD, ICO, DC, BNR, MAG, LIST	
Reviewers:	Constantinos Patsakis – ARC Alessandro Brighente - UNIPD	
Dissemination Level:	Public	X
	Confidential – only consortium members and European Commission Services	

This project has received financial support from the European Union Horizon 2020 Programme under grant agreement no. 101070303.

Revision History

Revision	Date	Author	Organization	Description
0.1	02/03/2023	Miltiadis Anastasiadis	MOT	TOC
0.2	10/03/2023	Nikos Stratigakis, Panagiotis Markovits, Miltiadis Anastasiadis	MOT	First Draft
0.3	20/03/2023	Andrei Costin, Vadim Bogulean, Jordan Atalianis, Lelia Ataliani, Thanos Karantjias, Spyridon Papastergiou, Tareq Chihabi, Nikos Drosos, Mirko Fabbri, Alessandro Neri, Mauro Scarpa, Nikolaos Karaiskakis, Filippo Paganelli, Nikos Stratigakis, Panagiotis Markovits, Miltiadis Anastasiadis	MAG, BNR, ICO, MOT	Updated with functional and non-functional user requirements
0.4	30/03/2023	Ana Lucila Sandoral, Sandra Perez, Luis Alberto Martinez, Fran Casino, Constantinos Patsakis, Mauro Conti, Alessandro Brighente, Yuejun Guo	WP3 partners, UNIPD, LIST, UCM, ARC	Updating user functional requirements
0.5	07/03/2023	Andrei Costin, Vadim Bogulean,	MAG, BNR, ICO, MOT	Updating overall user requirements

		Jordan Atalianis, Lelia Ataliani, Thanos Karantjias, Spyridon Papastergiou, Tareq Chihabi, Nikos Drosos, Mirko Fabbri, Alessandro Neri, Mauro Scarpa, Nikolaos Karaiskakis, Filippo Paganelli, Nikos Stratigakis, Panagiotis Markovits, Miltiadis Anastasiadis		
0.6	17/04/2023	Nikos Stratigakis, Panagiotis Markovits, Miltiadis Anastasiadis	MOT	Updating entire document with comments from all involved partners
0.8	25/04/2023	Constantinos Patsakis, Alessandro Brighente	ARC, UNIPD	Final internal review
1.0	30/04/2023	Nikos Stratigakis, Panagiotis Markovits, Miltiadis Anastasiadis		Final version

Every effort has been made to ensure that all statements and information contained herein are accurate, however the LAZARUS Project Partners accept no liability for any error or omission in the same.

Table of Contents

1 Executive Summary ... 6
2 Introduction ... 7
2.1 Structure of the document ... 7
2.2 Overview of LAZARUS Consolidated Use Cases ... 8
3 Methodology used ... 9
3.1 Methodology for requirements collection and analysis ... 10
3.2 End user requirements ... 11
3.3 Labelling and presentation of user requirements ... 12
4 Functional Requirements ... 20
4.1 Services ... 20
4.2 Expected Behaviour ... 25
5 Non-functional Requirements ... 28
5.1 Timing Constraints ... 28
5.2 Development Constraints ... 28
5.3 Standard Constraints ... 29
6 MoSCoW Requirements Analysis ... 31
7 WP3 Input ... 38
8 Consolidated User Requirements ... 41
8.1 General Requirements ... 41
8.2 USE CASE 1 - Issue detection regarding secrets management ... 46
8.3 USE CASE 2 - Code Linting ... 47
8.4 USE CASE 3 - Static Code Analysis ... 48
8.5 USE CASE 4 - SQL Injection Vulnerability Detection ... 49
8.6 USE CASE 5 - Fuzzing ... 50
8.7 USE CASE 6 - CVE Scan ... 50
8.8 USE CASE 7 - Container Vulnerability Scanning ... 52
8.9 USE CASE 8 - Detection of Network Attacks & DDoS ... 52
9 Conclusions ... 54
10 References ... 56

List of definitions & abbreviations

Abbreviation	Definition
API	Application Programming Interface
IDS	Intrusion Detection System
SME	Small Medium Enterprise
UC	Use Case
CVE	Common Vulnerabilities and Exposures
DDoS	Distributed Denial-of-Service
SDLC	Software Development Life Cycle
CFG	Control Flow Graph
EOL/EOS	end-of-life/end-of-sale
SBOM	Software Bill of Materials
DAST	Dynamic Application Security Testing

1 Executive Summary

In the given task, the primary objective is to gather end-user requirements from various use cases. The process involves conducting a comprehensive requirement analysis, which consists of several steps:

1. Interviews/Surveys with End Users and Key Stakeholders: To understand the expectations and needs of the people who will use the system, it is essential to communicate with them directly. This step involves conducting interviews or surveys with end users and key stakeholders to gather information about their preferences, concerns, and any specific requirements they may have.
2. Definition of User Requirements: After collecting feedback from end users and stakeholders, the next step is to compile and categorize the user requirements. This involves analyzing the collected data, identifying patterns, and defining clear and concise user requirements that reflect the needs and expectations of the target audience.
3. Step 2 above is enhanced and user requirements refined, by bringing onboard state of the art input from research organizations within the consortium,
4. Identification of Non-Functional Requirements: Non-functional requirements are aspects of the system that do not directly relate to its functionality but are essential for overall user satisfaction. Examples include performance, security, and usability. In this step, non-functional requirements are identified and documented to ensure the system meets these expectations.
5. Identification of Functional Requirements: Functional requirements describe the specific features and capabilities of the system. These requirements outline what the system is supposed to do and are essential for meeting user expectations. In this phase, the team identifies and documents the functional requirements based on user feedback and use case analysis.

Once the user requirements have been gathered and analyzed, the next step is to translate them into system requirements (Task 2.3). System requirements are a more technical description of the requirements and are used by developers to build the system. Both user requirements and system requirements serve as the foundation for work in Work Package 4 (WP4) and Work Package 5 (WP5), where the system will be designed, developed, and tested.

Finally, after the testing and validation (T&V) phase is completed, the requirements may be updated to address any issues or shortcomings identified during this process. This ensures that the system evolves and improves in response to user feedback and real-world performance, resulting in a more effective and satisfactory end product.

2 Introduction

D2.2 is a continuation of D2.1 that aims to identify and set functional and non-functional user requirements for the LAZARUS platform, both in general and in the context of specific use cases. In addition, the deliverable serves as the establishment of a communication channel between

- The pilots hosting partners to help them in describing in an accurate way their needs or wishes
- The research community and the pilot partners, through proposals, WP3 has made, for both user requirements and technological tools to be used for the Lazarus platform as an added value.
- The technology partners by supporting them in understanding exactly the mentioned needs or wishes

D2.2 - a very important deliverable towards designing a concise, realistic and market-oriented system - fully supports the LAZARUS stakeholders by proposing a standard procedure for requirements specification that will be used along the project lifetime for the development of the necessary components and interfaces.

2.1 Structure of the document

The deliverable is divided into the following chapters:

1. **Executive Summary** – condensed information summarizing the deliverable contents
2. **Introduction** – a short introduction of the deliverable goals
3. **Methodology used** – a high level presentation of the user requirements of the project, along with a description of the methodology used to extract them
4. **Functional Requirements** – a breakdown of the identified functional requirements into specific subcategories
5. **Non-functional Requirements** – a breakdown of the identified non-functional requirements into specific subcategories
6. **MoSCoW Requirements Analysis** – a presentation of the priorities set for the various user requirements, based on the MoSCoW prioritization technique
7. **WP3 Input –** a mapping showcasing in what ways WP3 will contributing towards the satisfaction of each user requirement

8. **Consolidated User Requirements –** a detailed analysis of all user requirements presented in previous sections

9. **Conclusion –** conclusion of the deliverable based on previous chapters and outline of next steps

10. **References**

2.2 Overview of LAZARUS Consolidated Use Cases

Herewith and for purposes of clarity and continuity within WP2, we present the current overview of all the use cases combined since the user requirements are formed per user case. This section provides a review of definitions of the use cases and acts as a link also between D2.1 and D2.2.

Use Case ID	High-Level Use Case Title	Domain
UC-1	Issue detection regarding secrets management	Pre-commit - Secrets Management
UC-2	Code Linting	Pre-commit – Code Linting
UC-3	Static Code Analysis	Vulnerability Scanning
UC-4	SQL Injection Vulnerability Detection	Vulnerability Scanning
UC-5	Fuzzing	Vulnerability Scanning
UC-6	CVE Scan	Vulnerability Scanning

UC-7	Container Vulnerability Scanning	Vulnerability Scanning
UC-8	Detection of Network Attacks & DDoS	Vulnerability Scanning

Table 2.1: Consolidated LAZARUS Use Cases

3 Methodology used

There are several definitions of what a requirement is. For LAZARUS **we agreed to use the definition of ISO (ISO/IEC 2007)**:

“A requirement is Statement that identifies a product (includes product, service, or enterprise) or process operational, functional, or design characteristic or constraint, which is unambiguous, testable or measurable, and necessary for product or process acceptability.”

Moreover, the term user requirements, in a specific technical sense, is the expression of the needs of the stakeholders in the utilization domain or how a user will interact with a system and what that user expects. In that sense it is really important to know the need of the people who are going to use the system.

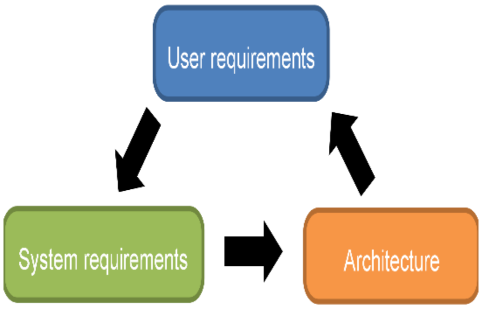


Figure 3.1: Systems Engineering development cycle

The project is following a Systems Engineering [3] approach to assist in ensuring the LAZARUS solution is suited to the practitioner-stakeholders for whom it is intended. The philosophy behind the approach and the role of the Stakeholders Requirements is illustrated in the Figure 1.

In the light of design experience, it is usual for the implementers to extract features and functionalities from the user requirements and constrains. However, implementers need to fully match user requirements hence the above presented feedback loop may not be required or may be repeated more than once.

3.1 Methodology for requirements collection and analysis

To achieve an extensive array of realistic and relevant requirements, both external and internal sources were used during the collection phase. Namely, with the pilot applications, defined use cases and consortium end user needs as a base, the requirements were enriched by input from the research partners (WP3) as well as the information received from the DevSecOps questionnaire responses (D2.1). The resulting requirement list was then processed and ordered according to their priority in relevance to the project goals. In a nutshell, the way user requirements are defined in LAZARUS, is depicted in the figure below.

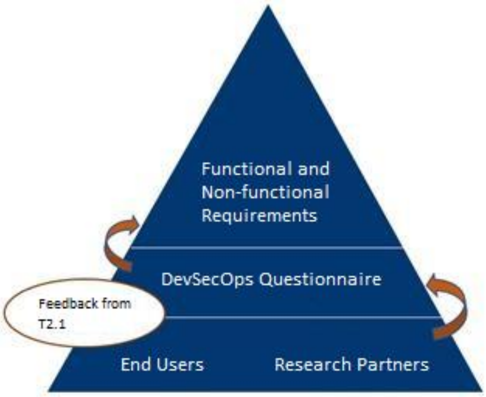


Figure 3.2: Process of requirement specification

Three main requirements ’sources are considered, comprising the LAZARUS extended stakeholder community:

- The first source reflects the End Users themselves that are part of the consortium, and which will host the pilots.
- The second source comprises the results of the state-of-the-art analysis in addition to the output gathered from the questionnaires prepared and have been kindly answered by external organizations that accepted to collaborate with the LAZARUS consortium.
- The third source is from input gathered from WP3 research organizations, who, through several long sometime meetings, provided valuable information and insights for both user requirements and tools to be provided as output of WP3.

The collected requirements are prioritized using the Moscow methodology, after taking into account the real needs of the LAZARUS extended community and stakeholders and those functionalities that are of primary importance and can be supported in designing and building the LAZARUS integrated platform.

3.2 End user requirements

End user requirements specify exactly what the software must do and describe the expectations of the user from a software that will be developed. As a part of the contractual agreement, the user requirement protects the developer from demands of a user for features that are not documented or non-contractual and prevents developers of claiming a software to be ready if it not fulfils the requirements. In the scope of information technology, end user requirements are used to clarify for whom an IT software product is developed. The term "end user" determines who will benefit from the developed product and who will finally use it. It distinguishes the user from other possible actors during a development process as e.g., administrators or system operators. User requirements analysis within LAZARUS include the following characteristics:

- are verifiable, clear and concise, complete, consistent, traceable, viable, necessary and implementation manageable,
- are precise and well-defined,
- are unique and not lengthy based on consortium experience, and,
- do not contain unnecessary definitions, are unambiguous and easy to read.

To move forward with the LAZARUS system design and architecture, the requirements captured through the methodology described above are classified as functional or non-functional. In a nutshell, functional requirements describe how the system should function from the user perspective. Non-functional requirements do not describe the functionality of the system, but they deal with other characteristics of the system such as performance, reliability, software quality, and cost, which are concerns for the stakeholders as well.

Requirement engineering differs between functional and non-functional requirements. End-users' requirements are divided into two main categories: functional and non-functional. Defining functional and non-functional requirements in a project is important but it's essential for a project that both types of requirements are fully taken into account during the development process.

- **Functional Requirements:** a functional requirement (FR) [1] defines a function of a system or its component, where a function is described as a specification of behaviour between inputs and outputs. In other words, functional requirements are LAZARUS platform features or functions that the developers must implement to enable users to accomplish their tasks. Functional requirements usually cover among others the following aspects: authorization levels, authentication, external interfaces, reporting, historical data, administrative functionality, and legal requirements.
- **Non-functional Requirements:** a non-functional requirement (NFR) [2] is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviours. They are contrasted with functional requirements that define specific behaviour or functions. Non-functional requirements are based on the quality of how a required functionality is provided. Quality in that sense can be how a functionality is performed or the conditions a functionality has to fulfil

covering among others the following aspects: performance, scalability, capacity, availability, maintainability, security, manageability, data integrity, usability, interoperability, recoverability, environmental and regulatory. The non-functional requirements are particularly important for the customer acceptance and therefore have to be defined with high attention.

Figure 3.3: Functional Requirements vs. Functional Requirements

3.3 Labelling and presentation of user requirements

At this point, the user requirements stemming from the use cases are a definition of requirements that are taken into account in the current user requirements deliverable. Each user requirement is labelled and analysed as per the table below content structure.

Requirement Label	Indicator	Description
UR-C-N-1	Compliance with existing security standards	Compliance with existing security standards (such as ISO27001, ISO 27002, ISO 27005, ISO 27035) [4] associated with the protection of the HealthCare/Energy/Transportation operators, mandated by law and regulation for the protection of critical infrastructures (NIS Directive, Directive 2002/21/EC [5], Directive (EU) 2016/1148 [6])
UR-C-F-1	Automated Compliance Checks	LAZARUS should automate compliance checks throughout the development life cycle, so as to identify and remediate potential compliance issues early in the development process, ensuring that

		applications, infrastructure, and configurations adhere to relevant security laws, regulations, and industry standards.
UR-C-F-2	Standards-based Policy Enforcement	LAZARUS should enable organizations to define and enforce policies based on the mentioned ISO standards (ISO 27001, ISO 27002, ISO 27005, and ISO 27035). By incorporating these standards into the development pipeline, organizations can ensure compliance with best practices and regulatory requirements.
UR-C-F-3	Mapping to Regulatory Frameworks	LAZARUS should have the capability to map security controls and requirements to specific laws and regulations, such as the NIS Directive, Directive 2002/21/EC, and Directive (EU) 2016/1148. This simplifies the compliance process and helps organizations demonstrate their adherence to the relevant legal frameworks.
UR-C-F-4	Risk Management	In line with the ISO 27005 standard, the tool should facilitate risk management by providing a framework for identifying, assessing, and managing information security risks throughout the development life cycle.
UR-CE-N-1	Scalability and Adaptability	The LAZARUS system should be modular and flexible. As security laws and regulations evolve, a DevSecOps tool should be scalable and adaptable to accommodate new requirements and standards. This ensures that organizations can maintain compliance without significant disruptions to their development processes.
UR-CE-N-2	Portability	The LAZARUS system should be portable (i.e., run on diverse operating systems) and able to replicate and be deployed across different infrastructures with low effort.
UR-C-F-5	Incident Management	As per the ISO 27035 standard, the LAZARUS system should support incident management capabilities, including preparing for, identifying, assessing, responding to, and learning from information security incidents. This can help organizations

		minimize the impact of security incidents and ensure timely recovery.
UR-C-F-6	Training and Awareness	A DevSecOps tool can include training and awareness modules to help educate developers and other stakeholders on security best practices and the importance of compliance. This can help foster a security-conscious culture and reduce the likelihood of security incidents due to human error.
UR-C-F-7	Authentication	The LAZARUS system shall require users to authenticate themselves before accessing any of its modules.
UR-C-F-8	Authorization	The LAZARUS system shall implement authorization to control access to its modules. Only authorized users shall be allowed to access the modules they are authorized to access.
UR-C-F-9	Role-based access control	LAZARUS should implement role-based access control to ensure that users can only access the modules that are relevant to their role.
UR-C-F-10	Module-specific access	LAZARUS could enforce access controls on a per-module basis to ensure that users can only access the modules they are authorized to access.
UR-C-F-11	Administrative Interfaces	LAZARUS should provide administrative interfaces for managing user accounts, roles, and module-level access controls. The interfaces shall enable administrators to create, modify, and delete user accounts, assign roles, and grant module-level access permissions to users and roles.
UR-C-F-12	Auditability	The LAZARUS system should maintain an audit trail of all authentication and authorization events, including login attempts, module accesses, and any changes made to user accounts, roles, or module-level access controls. The audit trail shall be accessible only to authorized users and shall be protected against unauthorized access, modification, or deletion.

UR-C-F-13	Verifiability	All data generated by LAZARUS must be verifiable
UR-C-F-14	Documentation and Reporting	The LAZARUS platform should support comprehensive documentation and reporting capabilities, enabling organizations to demonstrate their compliance with the mentioned security standards and regulations. This may include generating audit-ready reports, tracking remediation efforts, and providing evidence of security controls and risk management practices.
UR-C-F-15	Machine-accessible Reporting	The reporting provided by LAZARUS should provide the option to output any report in a standard machine-readable format (e.g., JSON, XML) so it can be parsed by an automated tool/dashboard.
UR-C-N-2	Source Code Confidentiality	Modules that have access to a repository must use the repository source code solely for the purpose of their functionality and must not export, send, or otherwise use any external tools or services that would expose the source code outside of the system without explicit authorization. Any use of the source code must be restricted to the specific context and scope of the module, and the source code must not be exposed to unauthorized users or systems.
UR-C-N-3	Efficient Processing	Avoid long-running processes in modules provided by LAZARUS and commonly used in SDLC “software development life cycle” (i.e., shouldn't be considerably longer than a pipeline or workflow without LAZARUS integration, such as security won’t slow down the checks or force the scan to be performed in asynchronous way)
UR-C-N-4	System Stability	The system shall provide a fail-safe configuration, i.e., in case of an unexpected event or error, the system shall go to a safe state.
UR-C-N-5	Multi-tenancy	When integrating resources that are used by multiple tenants/users (e.g., cloud environment), those shared resources shall support tenant separation / process isolation.

UR-C-F-16	Reliable Hardcoded Secret Detection	The LAZARUS system should be able to reliably identify hardcoded secrets in provided source code
UR-C-F-17	Reliable Unencrypted Secret Detection	The LAZARUS system should be able to reliably identify unencrypted secrets in provided source code
UR-C-F-18	Reliable Stored Secret Detection	The LAZARUS system should be able to reliably identify stored secrets in provided source code
UR-C-F-19	Secret Detection in Code History	The LAZARUS system could identify whether code history contains inadvertent secrets
UR-CE-F-1	Reliable Vulnerability Detection in Source Code	The LAZARUS system should be able to reliably detect errors in provided source code and indicate whether they can lead to security vulnerabilities
UR-C-F-20	Formatting and Styling Issue Detection	The LAZARUS system should be able to detect formatting or styling issues in provided source code
UR-C-F-21	Coding and Deployment Best Practices Suggestions	The LAZARUS system could suggest best practices based on provided source code, as well as best practice tips for software and hardware deployment.
UR-C-F-22	Source Code Pattern-based Simulation	The LAZARUS system could offer pattern-based simulation based on provided source code
UR-CE-F-2	Source Code Quality and Complexity Metrics	The LAZARUS system could offer quality and complexity metrics based on provided source code
UR-CE-F-3	Safety and Security Coding Standards Support	The LAZARUS system could support multiple safety and security-focused coding standards
UR-C-F-23	Out-of-the-box Certification	The LAZARUS system could support out-of-the-box certification for use in the development of safety-critical applications

UR-CE-F-4	Source Code-based Data Flow Analysis	The LAZARUS system must be able to offer data flow analysis based on provided source code
UR-CE-F-5	Source Code-based Control Flow Graphs	The LAZARUS system must be able to create Control Flow Graphs (CFG) based on provided source code
UR-C-F-24	Source Code-based Taint Analysis	The LAZARUS system must be able to offer taint analysis based on provided source code
UR-C-F-25	Source Code-based Lexical Analysis	The LAZARUS system must be able to offer lexical analysis based on provided source code
UR-C-F-26	Cryptography-related Issue Detection	Check for misused cryptographic functions, encryption/decryption modes, and Initialization Vector selection/handling (e.g., improper use of cryptographic primitives offered by the platform, using outdated algorithms as MD5 or bad practices with cyphers such as ECB, storing the same IV for every connection, etc.)
UR-CE-F-6	DAST-based SQL Injection Vulnerability Detection	The LAZARUS system must offer penetration testing services to detect possible injections at the API level
UR-CE-F-7	Query Input Whitelisting Verification	The LAZARUS system should be able to detect whether the provided source code whitelists query input validation
UR-CE-F-8	Query Input Escape Verification	The LAZARUS system should be able to detect whether the provided source code escapes all supplied query input
UR-CE-F-9	Fuzzing Service Configurability	The LAZARUS fuzzing services should be fully configurable, with the options to specify target(s), fuzzer(s), test cases, credentials, input types and combination logic
UR-C-F-27	Protocol Fuzzing Services	The LAZARUS system could offer protocol fuzzing services

UR-C-F-28	File Format Fuzzing Services	The LAZARUS system could offer file format fuzzing services
UR-CE-F-10	CVE Scanning Reliability	The LAZARUS system must be able to reliably detect outdated, end-of-life (EOL) and vulnerable components based on a provided Software Bill of Materials (SBOM)
UR-CE-F-11	CVE Scanning Service Accessibility	The LAZARUS system must be able to read the most widely used SBOM file formats (SPDX, CycloneDX, SWID, NPM package lock, Maven POM, etc.)
UR-CE-F-12	CVE Scanning Service Configurability	The LAZARUS system should provide a configurable policy list when scanning a Software Bill of Materials (SBOM), such as - Restrictions on component age - Restrictions on outdated and EOL/EOS components - Prohibition of components with known vulnerabilities - Restrictions on public repository usage - Restrictions on acceptable licenses - Component update requirements - Deny list of prohibited components and versions - Acceptable community contribution guidelines
UR-C-F-29	Unnecessary Direct and Transitive Dependencies Detection	The LAZARUS system could offer the option to detect unnecessary (unused) direct and transitive dependencies based on a provided Software Bill of Materials (SBOM)
UR-C-F-30	Project Dependencies Health Check	The LAZARUS system could offer the option to assess the health of project dependencies based on a provided Software Bill of Materials (SBOM)
UR-CE-F-13	Container Vulnerability Scanning Reliability	The LAZARUS system must be able to reliably detect insecure containers (outdated libraries, incorrectly configured containers, outdated operating system) based on a provided container image

UR-C-F-31	Container-based Compliance Validation Detection	The LAZARUS system should be able to reliably detect possible compliance validations based on a provided container image
UR-C-F-32	Container-related Best Practice Suggestions	The LAZARUS system could suggest best practices based on a provided container image
UR-CE-F-14	Network Tool Reliability	The LAZARUS system must be able to reliably detect vulnerabilities in tested IDS and/or networks
UR-CE-F-15	Network Tool Configurability	The LAZARUS system should provide detailed configuration options for IDS/network vulnerability checks
UR-C-F-33	Incident Response and Recovery Policy Suggestions	LAZARUS could enable organizations to develop and implement incident response and recovery plans as required by the regulations of their industry, through improvement and best practice suggestions.

Table 3.1: List of LAZARUS User Requirements

User requirements are labelled as ***UR-<type>-<section>-<number>***. Type can be E (external), C (consortium) or both (CE). External user requirements are provided by external stakeholders, while Consortium user requirements have been identified through internal development process by both end users and research partners in the LAZARUS Consortium. Section stands for type of the user requirements (functional or non-functional) and Number for the number of the user requirement the specific type. An example of such label is UR-E-F-2 which is an external user requirement of functional type and with user requirement number 2. This way it is easier for a reader to quickly refer to the source of the user requirements.

4 Functional Requirements

This section covers

- services (either general or regarding specific use cases), and
- how the LAZARUS system should behave in particular situations.

4.1 Services

As it is envisioned that each LAZARUS use case will provide a particular service, it is important to define the goals and features of the service in question. Thus, a breakdown of related requirements is necessary, ranging from mandatory core functions to nice-to-have optional features. In essence, the satisfaction of these requirements will comprise the core business logic of the platform.

Requirement Label	Indicator	Description	Use Case
UR-C-F-16	Reliable Hardcoded Secret Detection	The LAZARUS system should be able to reliably identify hardcoded secrets in provided source code	USE CASE 1 - Issue detection regarding secrets management
UR-C-F-17	Reliable Unencrypted Secret Detection	The LAZARUS system should be able to reliably identify unencrypted secrets in provided source code	USE CASE 1 - Issue detection regarding secrets management
UR-C-F-18	Reliable Stored Secret Detection	The LAZARUS system should be able to reliably identify stored secrets in provided source code	USE CASE 1 - Issue detection regarding secrets management
UR-C-F-19	Secret Detection in Code History	The LAZARUS system could identify whether code history contains inadvertent secrets	USE CASE 1 - Issue detection regarding secrets management
UR-CE-F-1	Reliable Vulnerability Detection in Source Code	The LAZARUS system should be able to reliably detect errors in provided source code and	USE CASE 2 - Code Linting

		indicate whether they can lead to security vulnerabilities	
UR-C-F-20	Formatting and Styling Issue Detection	The LAZARUS system should be able to detect formatting or styling issues in provided source code	USE CASE 2 - Code Linting
UR-C-F-21	Coding and Deployment Best Practices Suggestions	The LAZARUS system could suggest best practices based on provided source code, as well as best practice tips for software and hardware deployment.	USE CASE 2 - Code Linting
UR-C-F-22	Source Code Pattern-based Simulation	The LAZARUS system could offer pattern-based simulation based on provided source code	USE CASE 2 - Code Linting
UR-CE-F-2	Source Code Quality and Complexity Metrics	The LAZARUS system could offer quality and complexity metrics based on provided source code	USE CASE 2 - Code Linting
UR-CE-F-3	Safety and Security Coding Standards Support	The LAZARUS system could support multiple safety and security-focused coding standards	USE CASE 2 - Code Linting
UR-C-F-23	Out-of-the-box Certification	The LAZARUS system could support out-of-the-box certification for use in the development of safety-critical applications	USE CASE 2 - Code Linting
UR-CE-F-4	Source Code-based Data Flow Analysis	The LAZARUS system must be able to offer data flow analysis based on provided source code	USE CASE 3 - Static Code Analysis
UR-CE-F-5	Source Code-based Control Flow Graphs	The LAZARUS system must be able to create Control Flow Graphs (CFG) based on provided source code	USE CASE 3 - Static Code Analysis

UR-C-F-24	Source Code-based Taint Analysis	The LAZARUS system must be able to offer taint analysis based on provided source code	USE CASE 3 - Static Code Analysis
UR-C-F-25	Source Code-based Lexical Analysis	The LAZARUS system must be able to offer lexical analysis based on provided source code	USE CASE 3 - Static Code Analysis
UR-C-F-26	Cryptography-related Issue Detection	Check for misused cryptographic functions, encryption/decryption modes, and Initialization Vector selection/handling (e.g., improper use of cryptographic primitives offered by the platform, using outdated algorithms as MD5 or bad practices with cyphers such as ECB, storing the same IV for every connection, etc.)	USE CASE 3 - Static Code Analysis
UR-CE-F-6	DAST-based SQL Injection Vulnerability Detection	The LAZARUS system must offer penetration testing services to detect possible injections at the API level	USE CASE 4 - SQL Injection Vulnerability Detection
UR-CE-F-7	Query Input Whitelisting Verification	The LAZARUS system should be able to detect whether the provided source code whitelists query input validation	USE CASE 4 - SQL Injection Vulnerability Detection
UR-CE-F-8	Query Input Escape Verification	The LAZARUS system should be able to detect whether the provided source code escapes all supplied query input	USE CASE 4 - SQL Injection Vulnerability Detection
UR-CE-F-9	Fuzzing Service Configurability	The LAZARUS fuzzing services should be fully configurable, with the options to specify target(s), fuzzer(s), test cases, credentials, input types and combination logic	USE CASE 5 - Fuzzing
UR-C-F-27	Protocol Fuzzing Services	The LAZARUS system could offer protocol fuzzing services	USE CASE 5 - Fuzzing

UR-C-F-28	File Format Fuzzing Services	The LAZARUS system could offer file format fuzzing services	USE CASE 5 - Fuzzing
UR-CE-F-10	CVE Scanning Reliability	The LAZARUS system must be able to reliably detect outdated, End-Of-Life (EOL) and vulnerable components based on a provided Software Bill of Materials (SBOM)	USE CASE 6 - CVE Scan
UR-CE-F-11	CVE Scanning Service Accessibility	The LAZARUS system must be able to read the most widely used SBOM file formats (SPDX, CycloneDX, SWID, NPM package lock, Maven POM, etc.)	USE CASE 6 - CVE Scan
UR-CE-F-12	CVE Scanning Service Configurability	The LAZARUS system should provide a configurable policy list when scanning a Software Bill of Materials (SBOM), such as - Restrictions on component age - Restrictions on outdated and EOL/EOS components - Prohibition of components with known vulnerabilities - Restrictions on public repository usage - Restrictions on acceptable licenses - Component update requirements - Deny list of prohibited components and versions - Acceptable community contribution guidelines	USE CASE 6 - CVE Scan
UR-C-F-29	Unnecessary Direct and Transitive	The LAZARUS system could offer the option to detect unnecessary direct and transitive dependencies based on a	USE CASE 6 - CVE Scan

	Dependencies Detection	provided Software Bill of Materials (SBOM)	
UR-C-F-30	Project Dependencies Health Check	The LAZARUS system could offer the option to assess the health of project dependencies based on a provided Software Bill of Materials (SBOM)	USE CASE 6 - CVE Scan
UR-CE-F-13	Container Vulnerability Scanning Reliability	The LAZARUS system must be able to reliably detect insecure containers (outdated libraries, incorrectly configured containers, outdated operating system) based on a provided container image	USE CASE 7 - Container Vulnerability Scanning
UR-C-F-31	Container-based Compliance Validation Detection	The LAZARUS system should be able to reliably detect possible compliance validations based on a provided container image	USE CASE 7 - Container Vulnerability Scanning
UR-C-F-32	Container-related Best Practice Suggestions	The LAZARUS system could suggest best practices based on a provided container image	USE CASE 7 - Container Vulnerability Scanning
UR-CE-F-14	Network Tool Reliability	The LAZARUS system must be able to reliably detect vulnerabilities in tested IDS and/or networks	USE CASE 8 - Detection of Network Attacks & DDoS
UR-CE-F-15	Network Tool Configurability	The LAZARUS system should provide detailed configuration options for IDS/network vulnerability checks	USE CASE 8 - Detection of Network Attacks & DDoS
UR-C-F-33	Incident Response and Recovery Policy Suggestions	LAZARUS could enable organizations to develop and implement incident response and recovery plans as required by the regulations of their industry, through improvement and best practice suggestions.	USE CASE 8 - Detection of Network Attacks & DDoS

Table 4.1: List of LAZARUS Functional User Requirements - Services

4.2 Expected Behaviour

In order for the LAZARUS system to properly interact with its users and their environments, one should define the desired properties the system should have, as well as the effect it must achieve in relation to the development life cycle it interacts with. Such requirements essentially describe the inherent processes of the platform itself.

Requirement Label	Indicator	Description	Use Case
UR-C-F-1	Automated Compliance Checks	LAZARUS should automate compliance checks throughout the development life cycle, so as to identify and remediate potential compliance issues early in the development process, ensuring that applications, infrastructure, and configurations adhere to relevant security laws, regulations, and industry standards.	General
UR-C-F-2	Standards-based Policy Enforcement	LAZARUS should enable organizations to define and enforce policies based on the mentioned ISO standards (ISO 27001, ISO 27002, ISO 27005, and ISO 27035). By incorporating these standards into the development pipeline, organizations can ensure compliance with best practices and regulatory requirements.	General
UR-C-F-3	Mapping to Regulatory Frameworks	LAZARUS should have the capability to map security controls and requirements to specific laws and regulations, such as the NIS Directive, Directive 2002/21/EC, and Directive (EU) 2016/1148. This simplifies the compliance process and helps organizations demonstrate their adherence to the relevant legal frameworks.	General
UR-C-F-4	Risk Management	In line with the ISO 27005 standard, the tool should facilitate risk management by providing a framework for identifying, assessing, and managing information security risks throughout the development life cycle.	General

UR-C-F-5	Incident Management	As per the ISO 27035 standard, the LAZARUS system should support incident management capabilities, including preparing for, identifying, assessing, responding to, and learning from information security incidents. This can help organizations minimize the impact of security incidents and ensure timely recovery.	General
UR-C-F-6	Training and Awareness	A DevSecOps tool can include training and awareness modules to help educate developers and other stakeholders on security best practices and the importance of compliance. This can help foster a security-conscious culture and reduce the likelihood of security incidents due to human error.	General
UR-C-F-7	Authentication	The LAZARUS system shall require users to authenticate themselves before accessing any of its modules.	General
UR-C-F-8	Authorization	The LAZARUS system shall implement authorization to control access to its modules. Only authorized users shall be allowed to access the modules they are authorized to access.	General
UR-C-F-9	Role-based access control	LAZARUS should implement role-based access control to ensure that users can only access the modules that are relevant to their role.	General
UR-C-F-10	Module-specific access	LAZARUS could enforce access controls on a per-module basis to ensure that users can only access the modules they are authorized to access.	General
UR-C-F-11	Administrative Interfaces	LAZARUS should provide administrative interfaces for managing user accounts, roles, and module-level access controls. The interfaces shall enable administrators to create, modify, and delete user accounts, assign roles, and grant module-level access permissions to users and roles.	General
UR-C-F-12	Auditability	The LAZARUS system should maintain an audit trail of all authentication and authorization events, including login attempts, module accesses, and any changes made to user accounts, roles, or module-level access controls. The audit trail shall be accessible only to authorized users and shall be	General

		protected against unauthorized access, modification, or deletion.	
UR-C-F-13	Verifiability	All data generated by LAZARUS must be verifiable and reproducible.	General
UR-C-F-14	Documentation and Reporting	The LAZARUS platform should support comprehensive documentation and reporting capabilities, enabling organizations to demonstrate their compliance with the mentioned security standards and regulations. This may include generating audit-ready reports, tracking remediation efforts, and providing evidence of security controls and risk management practices.	General
UR-C-F-15	Machine-accessible Reporting	The reporting provided by LAZARUS should provide the option to output any report in a standard machine-readable format (JSON/XML) so it can be parsed by an automated tool/dashboard.	General

Table 4.2: List of LAZARUS Functional User Requirements – Expected Behavior

5 Non-functional Requirements

This section covers constraints on the services or functions offered by the LAZARUS system, such as

- timing constraints,
- constraints on the development process, and
- constraints imposed by standards.

5.1 Timing Constraints

It is apparent that each pilot application already has its own restrictions and constraints, depending on the execution environment, the particularities of its implementation and customer needs. Consequently, possible timing constraints (maximum accepted pilot downtime, maximum reaction time until LAZARUS is triggered, maximum execution time of the triggered function) must be taken into account.

Requirement Label	Indicator	Description
UR-C-N-3	Efficient Processing	Avoid long-running processes in modules provided by LAZARUS and commonly used in SDLC "software development life cycle" (i.e., should not be considerably longer than a pipeline or workflow without LAZARUS integration, such as security would not slow down the checks or force the scan to be performed in asynchronous way)

Table 5.1: List of LAZARUS Non-functional User Requirements – Timing Constraints

5.2 Development Constraints

Necessary limitations regarding the architecture, technologies and communication methods of LAZARUS so that it is compatible with the pilot applications are stated in this section. The end goal of this requirement category is to achieve a viable, accessible and interoperable system design.

Requirement Label	Indicator	Description

UR-CE-N-1	Scalability and Adaptability	The LAZARUS system should be modular and flexible. As security laws and regulations evolve, a DevSecOps tool should be scalable and adaptable to accommodate new requirements and standards. This ensures that organizations can maintain compliance without significant disruptions to their development processes.
UR-CE-N-2	Portability	The LAZARUS system should be portable (i.e., run on diverse operating systems) and able to replicate and be deployed across different infrastructures with low effort.
UR-C-N-2	Source Code Confidentiality	Modules that have access to a repository must use the repository source code solely for the purpose of their functionality and must not export, send, or otherwise use any external tools or services that would expose the source code outside of the system without explicit authorization. Any use of the source code must be restricted to the specific context and scope of the module, and the source code must not be exposed to unauthorized users or systems.
UR-C-N-4	System Stability	The system shall provide a fail-safe configuration. i.e. in case of an unexpected event or error, the system shall go to a safe state.
UR-C-N-5	Multi-tenancy	When integrating resources that are used by multiple tenants/users (e.g. cloud environment), those shared resources shall support tenant separation / process isolation.

Table 5.2: List of LAZARUS Non-functional User Requirements – Development Constraints

5.3 Standard Constraints

Possible constraints related to imposed standards that LAZARUS aims to comply to, are mentioned in this section.

Requirement Label	Indicator	Description
UR-C-N-1	Compliance with existing security standards	Compliance with existing security standards (such as ISO27001, ISO 27002, ISO 27005, ISO 27035) [4] associated with the protection of the HealthCare/Energy/Transportation operators,

		mandated by law and regulation for the protection of critical infrastructures (NIS Directive, Directive 2002/21/EC [5], Directive (EU) 2016/1148 [6])

Table 5.3: List of LAZARUS Non-functional User Requirements – Standard Constraints

6 MoSCoW Requirements Analysis

When implementing the functionality of a system, it is important to prioritize the requirements focusing on the first development of the essential parts and remove the less significant ones if necessary due to the lack of time or resources.

In LAZARUS, the requirements are ranked based on the initial stakeholders' needs, input by external stakeholders and input by research partners within the project, coupled with the expertise of the partners. It is necessary to prioritize what is essential for the operation of the product for the development. The prioritization technique used as a reference to classify the requirements is MoSCoW [7].

MoSCoW was developed by Dai Clegg of Oracle UK in 1994 and it gained popularity in the DSDM methodology (Dynamic Software Development Method). The MoSCoW method is a prioritization technique used in management, business analysis, project management, and software development to reach a common understanding with stakeholders on the importance they place on the delivery of each requirement - also known as MoSCoW prioritization or MoSCoW analysis.

MoSCoW is a fairly simple way to sort features into priority order – a way to help teams quickly understand from the customer's view what is essential for launching a product and what is not. The MoSCoW method is a prioritization technique used in management, business analysis, project management, and software development to reach a common understanding with stakeholders on the importance they place on the delivery of each requirement; it is also known as MoSCoW prioritization or MoSCoW analysis.

The term MOSCOW itself is an acronym derived from the first letter of each of four prioritization categories: M - Must have, S - Should have, C - Could have, W - Won't have.

Naturally, all requirements are important, however to deliver the greatest and most immediate business benefits early the requirements must be prioritized. Initially the goal is to try and deliver all the Must have, Should have, and Could have requirements but the Should and Could requirements will be the first to be removed if the delivery timescale looks threatened.

MoSCoW is often used with timeboxing, where a deadline is fixed so that the focus must be on the most important requirements, and is commonly used in agile software development approaches such as Scrum, rapid application development (RAD), and DSDM.

Category	Explanation
Must have	Non-negotiable product needs that are mandatory for the team
Should have	Important initiatives that are not vital, but add significant value
Could have	Nice to have initiatives that will have a small impact if left out
Will not have	Initiatives that are not priority for this specific time frame

Table 6.1: MoScoW Categories

Requirement Code	Indicator	Category	MoSCoW Priority
LZR-GR1	Compliance with existing security standards	Non-functional	MUST
LZR-GR2	Automated Compliance Checks	Functional	SHOULD
LZR-GR3	Standards-based Policy Enforcement	Functional	SHOULD
LZR-GR4	Mapping to Regulatory Frameworks	Functional	COULD
LZR-GR5	Risk Management	Functional	MUST
LZR-GR6	Scalability and Adaptability	Non-functional	MUST
LZR-GR7	Portability	Non-functional	SHOULD
LZR-GR8	Incident Management	Functional	MUST
LZR-GR9	Training and Awareness	Functional	COULD
LZR-GR10	Authentication	Functional	MUST

LZR-GR11	Authorization	Functional	MUST
LZR-GR12	Role-based access control	Functional	SHOULD
LZR-GR13	Module-specific access	Functional	COULD
LZR-GR14	Administrative Interfaces	Functional	SHOULD
LZR-GR15	Auditability	Functional	MUST
LZR-GR16	Verifiability	Functional	MUST
LZR-GR17	Documentation and Reporting	Functional	MUST
LZR-GR18	Machine-accessible Reporting	Functional	SHOULD
LZR-GR19	Source Code Confidentiality	Non-functional	MUST
LZR-GR20	Efficient Processing	Non-functional	MUST
LZR-GR21	System Stability	Non-functional	MUST

LZR-GR22	Multi-tenancy	Non-functional	MUST
LZR-SM1	Reliable Hardcoded Secret Detection	Functional	MUST
LZR-SM2	Reliable Unencrypted Secret Detection	Functional	MUST
LZR-SM3	Reliable Stored Secret Detection	Functional	MUST
LZR-SM4	Secret Detection in Code History	Functional	COULD
LZR-CL1	Reliable Vulnerability Detection in Source Code	Functional	MUST
LZR-CL2	Formatting and Styling Issue Detection	Functional	SHOULD
LZR-CL3	Coding and Deployment Best Practices Suggestions	Functional	COULD
LZR-CL4	Source Code Pattern-based Simulation	Functional	COULD
LZR-CL5	Source Code Quality and Complexity Metrics	Functional	COULD
LZR-CL6	Safety and Security Coding Standards Support	Functional	COULD

LZR-CL7	Out-of-the-box Certification	Functional	COULD
LZR-SA1	Source Code-based Data Flow Analysis	Functional	MUST
LZR-SA2	Source Code-based Control Flow Graphs	Functional	MUST
LZR-SA3	Source Code-based Taint Analysis	Functional	COULD
LZR-SA4	Source Code-based Lexical Analysis	Functional	COULD
LZR-SA5	Cryptography-related Issue Detection	Functional	COULD
LZR-SI1	DAST-based SQL Injection Vulnerability Detection	Functional	MUST
LZR-SI2	Query Input Whitelisting Verification	Functional	SHOULD
LZR-SI3	Query Input Escape Verification	Functional	SHOULD
LZR-FZ1	Fuzzing Service Configurability	Functional	SHOULD
LZR-FZ2	Protocol Fuzzing Services	Functional	COULD

LZR-FZ3	File Format Fuzzing Services	Functional	COULD
LZR-CV1	CVE Scanning Reliability	Functional	MUST
LZR-CV2	CVE Scanning Service Accessibility	Functional	MUST
LZR-CV3	CVE Scanning Service Configurability	Functional	SHOULD
LZR-CV4	Unnecessary Direct and Transitive Dependencies Detection	Functional	COULD
LZR-CV5	Project Dependencies Health Check	Functional	COULD
LZR-CS1	Container Vulnerability Scanning Reliability	Functional	MUST
LZR-CS2	Container-based Compliance Validation Detection	Functional	SHOULD
LZR-CS3	Container-related Best Practice Suggestions	Functional	COULD
LZR-NA1	Network Tool Reliability	Functional	MUST
LZR-NA2	Network Tool Configurability	Functional	SHOULD

LZR-NA3	Incident Response and Recovery Policy Suggestions	Functional	COULD

Table 6.2: LAZARUS MoScoW Requirement List

7 WP3 Input

Research Organization	Requirement Implementation Contributions
ARC	Task 3.2 studied tools to provide a set of functionalities to analyse the composition of a project and the vulnerabilities of their components, either libraries, dependencies, or code. Moreover, further code-analysis capabilities, namely code analysis to determine potential flaws, quality and sanitisation were also studied. In summary, the deliverable presents a series of suitable open-source tools, some of them to be potentially integrated during development phase. Further functionalities such as a sanitisation module will be assessed and provided by research partners (ARC, DC), which will find personal, private and other data that could present potential security flaws in projects. Other background and tools can be provided by other partners if deemed necessary. The input of these tools assumes a project or directly a standardised file input (i.e., see D3.2 for more on SBOM standardised formats and other naming schemes), to create either a standarized report of its contents, a vulnerability analysis, or a static analysis. Furthermore, code analysis capabilities are also available via some of the tools analysed in D3.2. We foresee one or multiple modules that will be used to process these inputs and update the project's activities accordingly. Further discussions are needed to: 1. Select the desired functionalities for LAZARUS in the context of D3.2 2. Select the open-source tools providing them 3. Establish an integration strategy which enables the seamless operations over a project and their auditability 4. Ensure the compatibility of this/these modules with the rest of the platform. In the case of D3.8, the main aim was to study the standardised formats and tools to share Cyber threat intelligence. The document provides the main sharing and analysis platforms and their main characteristics. Overall, in terms of LAZARUS requirements and capabilities, D3.8 servers as a first basis of discussion to select: 1. Select the desired functionalities for LAZARUS in the context of D3.8. That is, which type of analysis and reporting capabilities we want to provide, following the state of the art.

	2. Select which format(s) will be used, and if any, with which intelligence tools LAZARUS will have the ability to communicate. 3. Establish an integration strategy which enables the seamless analysis and communication capabilities of LAZARUS, and their auditability. 4. Ensure the compatibility of this modules with the rest of the platform. Although D3.8 provides the necessary information to leverage decisions, a discussion is needed between WP3 and WP4 partners to finally select which will be the strategy to follow and integrate into the LAZARUS platform.
UNIPD	Task 3.1 will provide the pipeline for product analysis and the identification of vulnerabilities. As an output, it will also provide a list of possible mitigation approaches. • Input: Software Bill of Material (SBOM), comprising code, modules, libraries, and (if needed) indications on used hardware components. The SBOM shall be provided in a standardized format (e.g., CycloneDX) • Output: battery of security tests derived from the input SBOM • Supported programming language: Python • Code granularity: function level • Type of vulnerabilities: based on use cases • Technical prerequisites for executing the program: depends on the type/size of the model.
UCM	Task 3.3. An algorithmic verification tool will be developed to handle complex features of programs related to their control structure or memory management. These technologies will combine automatic source code abstraction techniques, symbolic model checking or counterexample-guided refinement of abstraction. • Input: Source code or binary program • Output: Vulnerabilities detected in source code or binary programs • Supported programming language: Python • Code granularity: function level • Type of vulnerabilities: based on use cases • Technical prerequisites for executing the program: The requirements will depend on the size of the code and the binary program to be analysed. Task 3.4. A tool will be developed using deep learning models to assess the security, detect failures and determine the level of robustness of the Artificial Intelligence techniques implemented in tasks T3.2 and 3.3. • Input: Random input for generation of adversarial samples.

	• Output: Adversarial sample • Supported programming language: based on use cases • Code granularity: function level • Type of vulnerabilities: based on use cases • Technical prerequisites for executing the program: Sufficient computational resources to generate the adversary samples. AI models implemented in tasks 3.2 and 3.3. For white box testing, overview of the AI model implemented in tasks 3.2 and 3.3
LIST	Task 3.5 will provide two AI models for the self-healing module and anti-fuzzing module, respectively. The self-healing module will focus on automated program repair that generates patches to vulnerable programs automatically. In detail: • Input: a piece of vulnerable code (remark: the vulnerability type should be identified by the vulnerability detection module in T2.2.) • Output: a patched code • Supported programming language: Java, Python, and C • Code granularity: function level (the input code is a defined function in a certain language) • Type of vulnerabilities: TBD. It is impractical to build an AI model that can fix all types of vulnerabilities due to the difficulty in collecting such data and the computational cost. Thus, several types of vulnerabilities will be determined based on the user requirements. • AI model: the model will be built upon a pre-trained large language model with the SOTA performance in automated program repair. The anti-fuzzing module is about automatically identifying fuzzers. • Input: the execution state of an application program • Output: whether a fuzzer is attacking the program or not. If yes, identify the name of the fuzzer. • Supported programming language of applications: Java, Python, and C • Fuzzers to consider: about five known fuzzers. These fuzzers will be determined based on the user requirements or in the literature review (e.g., AFL, HonggFuzz, Fairfuzz, VUzzer). • AI model: the model will be built upon a pre-trained large language model with the SOTA performance in various software engineering downstream tasks.

Table 7.1: WP3 Contributions

8 Consolidated User Requirements

The consolidated user requirements governing LAZARUS are herewith presented. The requirements will be translated into system requirements (T2.3) while they will also be the basis of work in WP4 and WP5. The requirements will be updated after the completion of T&V phase to reflect issues that were identified in this process.

8.1 General Requirements

ID	LZR-GR1	MoSCoW Priority	MUST
Name	Compliance with existing security standards		
Description	Compliance with existing security standards (such as ISO27001, ISO 27002, ISO 27005, ISO 27035) [4] associated with the protection of the HealthCare/Energy/Transportation operators, mandated by law and regulation for the protection of critical infrastructures (NIS Directive, Directive 2002/21/EC [5], Directive (EU) 2016/1148 [6])		
Category	Non-functional		

ID	LZR-GR2	MoSCoW Priority	SHOULD
Name	Automated Compliance Checks		
Description	LAZARUS should automate compliance checks throughout the development life cycle, so as to identify and remediate potential compliance issues early in the development process, ensuring that applications, infrastructure, and configurations adhere to relevant security laws, regulations, and industry standards.		
Category	Functional		

ID	LZR-GR3	MoSCoW Priority	SHOULD
Name	Standards-based Policy Enforcement		
Description	LAZARUS should enable organizations to define and enforce policies based on the mentioned ISO standards (ISO 27001, ISO 27002, ISO 27005, and ISO 27035). By incorporating these standards into the development pipeline, organizations can ensure compliance with best practices and regulatory requirements.		
Category	Functional		

ID	LZR-GR4	MoSCoW Priority	COULD
Name	Mapping to Regulatory Frameworks		
Description	LAZARUS should have the capability to map security controls and requirements to specific laws and regulations, such as the NIS Directive, Directive 2002/21/EC, and Directive (EU) 2016/1148. This simplifies the compliance process and helps organizations demonstrate their adherence to the relevant legal frameworks.		
Category	Functional		

ID	LZR-GR5	MoSCoW Priority	MUST
Name	Risk Management		
Description	In line with the ISO 27005 standard, the tool should facilitate risk management by providing a framework for identifying, assessing, and managing information security risks throughout the development life cycle.		
Category	Functional		

ID	LZR-GR6	MoSCoW Priority	MUST
Name	Scalability and Adaptability		
Description	The LAZARUS system should be modular and flexible. As security laws and regulations evolve, a DevSecOps tool should be scalable and adaptable to accommodate new requirements and standards. This ensures that organizations can maintain compliance without significant disruptions to their development processes.		
Category	Non-functional		

ID	LZR-GR7	MoSCoW Priority	SHOULD
Name	Portability		
Description	The LAZARUS system should be portable (i.e. run on diverse operating systems) and able to replicate and be deployed across different infrastructures with low effort.		
Category	Non-functional		

ID	LZR-GR8	MoSCoW Priority	MUST
Name	Incident Management		

Description	As per the ISO 27035 standard, the LAZARUS system should support incident management capabilities, including preparing for, identifying, assessing, responding to, and learning from information security incidents. This can help organizations minimize the impact of security incidents and ensure timely recovery.
Category	Functional

ID	LZR-GR9	MoSCoW Priority	COULD
Name	Training and Awareness		
Description	A DevSecOps tool can include training and awareness modules to help educate developers and other stakeholders on security best practices and the importance of compliance. This can help foster a security-conscious culture and reduce the likelihood of security incidents due to human error.		
Category	Functional		

ID	LZR-GR10	MoSCoW Priority	MUST
Name	Authentication		
Description	The LAZARUS system shall require users to authenticate themselves before accessing any of its modules.		
Category	Functional		

ID	LZR-GR11	MoSCoW Priority	MUST
Name	Authorization		
Description	The LAZARUS system shall implement authorization to control access to its modules. Only authorized users shall be allowed to access the modules they are authorized to access.		
Category	Functional		

ID	LZR-GR12	MoSCoW Priority	SHOULD
Name	Role-based access control		
Description	LAZARUS should implement role-based access control to ensure that users can only access the modules that are relevant to their role.		
Category	Functional		

ID	LZR-GR13	MoSCoW Priority	COULD

Name	Module-specific access
Description	LAZARUS could enforce access controls on a per-module basis to ensure that users can only access the modules they are authorized to access.
Category	Functional

ID	LZR-GR14	MoSCoW Priority	SHOULD
Name	Administrative Interfaces		
Description	LAZARUS should provide administrative interfaces for managing user accounts, roles, and module-level access controls. The interfaces shall enable administrators to create, modify, and delete user accounts, assign roles, and grant module-level access permissions to users and roles		
Category	Functional		

ID	LZR-GR15	MoSCoW Priority	MUST
Name	Auditability		
Description	The LAZARUS system should maintain an audit trail of all authentication and authorization events, including login attempts, module accesses, and any changes made to user accounts, roles, or module-level access controls. The audit trail shall be accessible only to authorized users and shall be protected against unauthorized access, modification, or deletion.		
Category	Functional		

ID	LZR-GR16	MoSCoW Priority	MUST
Name	Verifiability		
Description	All data generated by LAZARUS must be verifiable		
Category	Functional		

ID	LZR-GR17	MoSCoW Priority	MUST
Name	Documentation and Reporting		
Description	The LAZARUS platform should support comprehensive documentation and reporting capabilities, enabling organizations to demonstrate their compliance with the mentioned security standards and regulations. This may include generating audit-ready reports, tracking remediation efforts, and providing evidence of security controls and risk management practices.		

Category	Functional

ID	LZR-GR18	MoSCoW Priority	SHOULD
Name	Machine-accessible Reporting		
Description	The reporting provided by LAZARUS should provide the option to output any report in a standard machine-readable format (JSON/XML) so it can be parsed by an automated tool/dashboard.		
Category	Functional		

ID	LZR-GR19	MoSCoW Priority	MUST
Name	Source Code Confidentiality		
Description	Modules that have access to a repository must use the repository source code solely for the purpose of their functionality and must not export, send, or otherwise use any external tools or services that would expose the source code outside of the system without explicit authorization. Any use of the source code must be restricted to the specific context and scope of the module, and the source code must not be exposed to unauthorized users or systems.		
Category	Non-functional		

ID	LZR-GR20	MoSCoW Priority	MUST
Name	Efficient Processing		
Description	Avoid long-running processes in modules provided by LAZARUS and commonly used in SDLC “software development life cycle” (i.e. shouldn't be considerably longer than a pipeline or workflow without LAZARUS integration, such as security won’t slow down the checks or force the scan to be performed in asynchronous way)		
Category	Non-functional		

ID	LZR-GR21	MoSCoW Priority	MUST
Name	System Stability		
Description	The system shall provide a fail-safe configuration. i.e. in case of an unexpected event or error, the system shall go to a safe state.		
Category	Non-functional		

ID	LZR-GR22	MoSCoW Priority	MUST
Name	Multi-tenancy		
Description	When integrating resources that are used by multiple tenants/users (e.g. cloud environment), those shared resources shall support tenant separation / process isolation.		
Category	Non-functional		

8.2 USE CASE 1 - Issue detection regarding secrets management

ID	LZR-SM1	MoSCoW Priority	MUST
Name	Reliable Hardcoded Secret Detection		
Description	The LAZARUS system should be able to reliably identify hardcoded secrets in provided source code		
Category	Functional		

ID	LZR-SM2	MoSCoW Priority	MUST
Name	Reliable Unencrypted Secret Detection		
Description	The LAZARUS system should be able to reliably identify unencrypted secrets in provided source code		
Category	Functional		

ID	LZR-SM3	MoSCoW Priority	MUST
Name	Reliable Stored Secret Detection		
Description	The LAZARUS system should be able to reliably identify stored secrets in provided source code		
Category	Functional		

ID	LZR-SM4	MoSCoW Priority	COULD
Name	Secret Detection in Code History		
Description	The LAZARUS system could identify whether code history contains inadvertent secrets		

Category	Functional

8.3 USE CASE 2 - Code Linting

ID	LZR-CL1	MoSCoW Priority	MUST
Name	Reliable Vulnerability Detection in Source Code		
Description	The LAZARUS system should be able to reliably detect errors in provided source code and indicate whether they can lead to security vulnerabilities		
Category	Functional		

ID	LZR-CL2	MoSCoW Priority	SHOULD
Name	Formatting and Styling Issue Detection		
Description	The LAZARUS system should be able to detect formatting or styling issues in provided source code		
Category	Functional		

ID	LZR-CL3	MoSCoW Priority	COULD
Name	Coding and Deployment Best Practices Suggestions		
Description	The LAZARUS system could suggest best practices based on provided source code, as well as best practice tips for software and hardware deployment.		
Category	Functional		

ID	LZR-CL4	MoSCoW Priority	COULD
Name	Source Code Pattern-based Simulation		
Description	The LAZARUS system could offer pattern-based simulation based on provided source code		
Category	Functional		

ID	LZR-CL5	MoSCoW Priority	COULD
Name	Source Code Quality and Complexity Metrics		
Description	The LAZARUS system could offer quality and complexity metrics based on provided source code		

Category	Functional

ID	LZR-CL6	MoSCoW Priority	COULD
Name	Safety and Security Coding Standards Support		
Description	The LAZARUS system could support multiple safety and security-focused coding standards		
Category	Functional		

ID	LZR-CL7	MoSCoW Priority	COULD
Name	Out-of-the-box Certification		
Description	The LAZARUS system could support out-of-the-box certification for use in the development of safety-critical applications		
Category	Functional		

8.4 USE CASE 3 - Static Code Analysis

ID	LZR-SA1	MoSCoW Priority	MUST
Name	Source Code-based Data Flow Analysis		
Description	The LAZARUS system must be able to offer data flow analysis based on provided source code		
Category	Functional		

ID	LZR-SA2	MoSCoW Priority	MUST
Name	Source Code-based Control Flow Graphs		
Description	The LAZARUS system must be able to create Control Flow Graphs (CFG) based on provided source code		
Category	Functional		

ID	LZR-SA3	MoSCoW Priority	COULD
Name	Source Code-based Taint Analysis		

Description	The LAZARUS system must be able to offer taint analysis based on provided source code
Category	Functional

ID	LZR-SA4	MoSCoW Priority	COULD
Name	Source Code-based Lexical Analysis		
Description	The LAZARUS system must be able to offer lexical analysis based on provided source code		
Category	Functional		

ID	LZR-SA5	MoSCoW Priority	COULD
Name	Cryptography-related Issue Detection		
Description	Check for misused cryptographic functions, encryption/decryption modes, and Initialization Vector selection/handling (e.g. improper use of cryptographic primitives offered by the platform, using outdated algorithms as MD5 or bad practices with cyphers such as ECB, storing the same IV for every connection, etc.)		
Category	Functional		

8.5 USE CASE 4 - SQL Injection Vulnerability Detection

ID	LZR-SI1	MoSCoW Priority	MUST
Name	DAST-based SQL Injection Vulnerability Detection		
Description	The LAZARUS system must offer penetration testing services to detect possible injections at the API level		
Category	Functional		

ID	LZR-SI2	MoSCoW Priority	SHOULD
Name	Query Input Whitelisting Verification		
Description	The LAZARUS system should be able to detect whether the provided source code whitelists query input validation		
Category	Functional		

ID	LZR-SI3	MoSCoW Priority	SHOULD
Name	Query Input Escape Verification		
Description	The LAZARUS system should be able to detect whether the provided source code escapes all supplied query input		
Category	Functional		

8.6 USE CASE 5 - Fuzzing

ID	LZR-FZ1	MoSCoW Priority	SHOULD
Name	Fuzzing Service Configurability		
Description	The LAZARUS fuzzing services should be fully configurable, with the options to specify target(s), fuzzer(s), test cases, credentials, input types and combination logic		
Category	Functional		

ID	LZR-FZ2	MoSCoW Priority	COULD
Name	Protocol Fuzzing Services		
Description	The LAZARUS system could offer protocol fuzzing services		
Category	Functional		

ID	LZR-FZ3	MoSCoW Priority	COULD
Name	File Format Fuzzing Services		
Description	The LAZARUS system could offer file format fuzzing services		
Category	Functional		

8.7 USE CASE 6 - CVE Scan

ID	LZR-CV1	MoSCoW Priority	MUST
Name	CVE Scanning Reliability		

Description	The LAZARUS system must be able to reliably detect outdated, end-of-life (EOL) and vulnerable components based on a provided Software Bill of Materials (SBOM)
Category	Functional

ID	LZR-CV2	MoSCoW Priority	MUST
Name	CVE Scanning Service Accessibility		
Description	The LAZARUS system must be able to read the most widely used SBOM file formats (SPDX, CycloneDX, SWID,NPM package lock, Maven POM, etc.)		
Category	Functional		

ID	LZR-CV3	MoSCoW Priority	SHOULD
Name	CVE Scanning Service Configurability		
Description	The LAZARUS system should provide a configurable policy list when scanning a Software Bill of Materials (SBOM), such as - Restrictions on component age - Restrictions on outdated and EOL/EOS components - Prohibition of components with known vulnerabilities - Restrictions on public repository usage - Restrictions on acceptable licenses - Component update requirements - Deny list of prohibited components and versions - Acceptable community contribution guidelines		
Category	Functional		

ID	LZR-CV4	MoSCoW Priority	COULD
Name	Unnecessary Direct and Transitive Dependencies Detection		
Description	The LAZARUS system could offer the option to detect unnecessary direct and transitive dependencies based on a provided Software Bill of Materials (SBOM)		
Category	Functional		

ID	LZR-CV5	MoSCoW Priority	COULD
Name	Project Dependencies Health Check		

Description	The LAZARUS system could offer the option to assess the health of project dependencies based on a provided Software Bill of Materials (SBOM)
Category	Functional

8.8 USE CASE 7 - Container Vulnerability Scanning

ID	LZR-CS1	MoSCoW Priority	MUST
Name	Container Vulnerability Scanning Reliability		
Description	The LAZARUS system must be able to reliably detect insecure containers (outdated libraries, incorrectly configured containers, outdated operating system) based on a provided container image		
Category	Functional		

ID	LZR-CS2	MoSCoW Priority	SHOULD
Name	Container-based Compliance Validation Detection		
Description	The LAZARUS system should be able to reliably detect possible compliance validations based on a provided container image		
Category	Functional		

ID	LZR-CS3	MoSCoW Priority	COULD
Name	Container-related Best Practice Suggestions		
Description	The LAZARUS system could suggest best practices based on a provided container image		
Category	Functional		

8.9 USE CASE 8 - Detection of Network Attacks & DDoS

ID	LZR-NA1	MoSCoW Priority	MUST
Name	Network Tool Reliability		
Description	The LAZARUS system must be able to reliably detect vulnerabilities in tested IDS and/or networks		
Category	Functional		

ID	LZR-NA2	MoSCoW Priority	SHOULD
Name	Network Tool Configurability		
Description	The LAZARUS system should provide detailed configuration options for IDS/network vulnerability checks		
Category	Functional		

ID	LZR-NA3	MoSCoW Priority	COULD
Name	Incident Response and Recovery Policy Suggestions		
Description	LAZARUS could enable organizations to develop and implement incident response and recovery plans as required by the regulations of their industry, through improvement and best practice suggestions.		
Category	Functional		

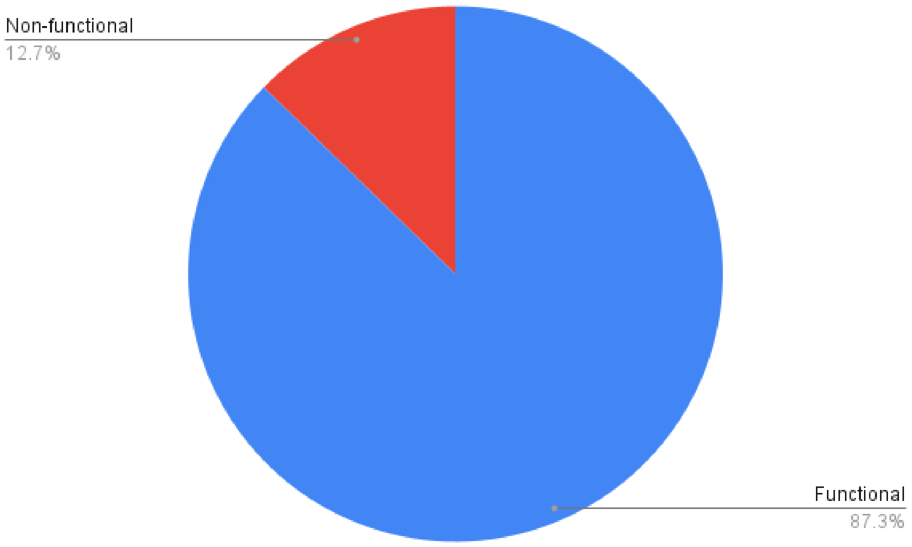
9 Conclusions

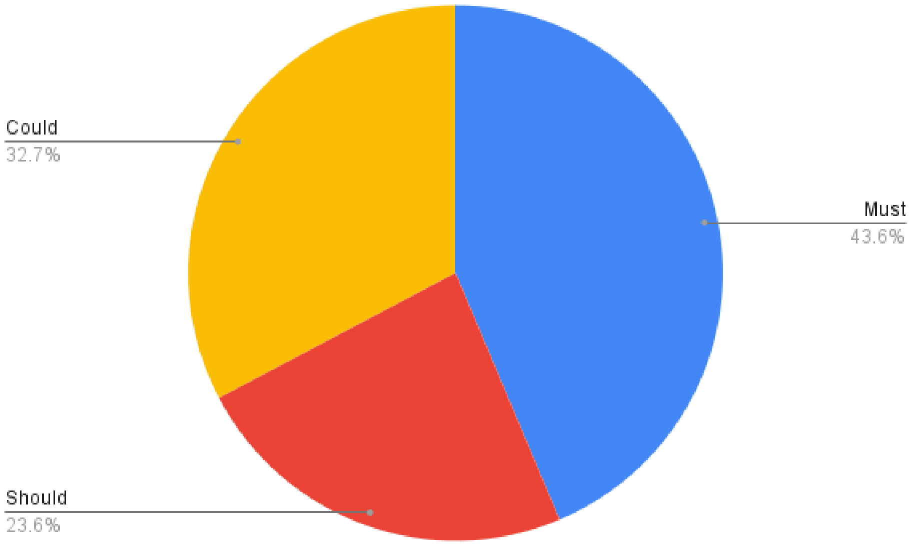
This deliverable provides the User Requirements for the components to be developed in LAZARUS project. The term "User Requirements" is used in a specific technical sense as "the expression of the needs of all stakeholders in the utilization domain", and the language used is that of the practitioners, describing their operational, functional and non-functional needs.

All user requirements and project goals that have been analysed in several iterations with users and partners have been taken into account in defining and streamlining system specifications per module and in an integrated way. In total 55 requirements have been identified and split as follows:

- Per category (
  - 48 functional
  - 7 non-functional

- Per priority
  - 24 MUST
  - 13 SHOULD
  - 18 COULD





The report describes the process used for eliciting the requirements, the methodology used to collect data from all stated sources, the analysis of the results, and the identification and prioritisation of the user requirements.

The final set of LAZARUS user requirements presented in the report will be the basis for LAZARUS system specifications (D2.3) and architecture design (D2.4) and development phases (WP4), although, it is an iterative process that will be reviewed during the development phase. Additionally, User Requirements will also be an inherent part of the evaluation process and of course of the dissemination and exploitation scenarios stating key functions of the system and its Unique Selling Points (USPs).

10 References

[1] https://en.wikipedia.org/wiki/Functional_requirement

[2] https://en.wikipedia.org/wiki/Non-functional_requirement

[3] https://en.wikipedia.org/wiki/Systems_engineering

[4] https://www.iso.org/standard/iso-iec-27000-family

[5] https://eur-lex.europa.eu/legal-content/en/ALL/?uri=CELEX%3A32002L0021

[6] https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex:32016L1148

[7] https://en.wikipedia.org/wiki/MoSCoW_method