An Expert Report on Fundamental Software Requirement Topics

I. Introduction to Software Requirements Engineering

A. The Critical Role of Requirements in Software Success

Software requirements are the foundational elements upon which successful software projects are built. They serve as the blueprint that guides the entire development and implementation lifecycle of a software solution.[1] The process of defining clear and comprehensive software requirements at the project's inception is paramount, as it sets the stage for aligning the final product with specific business needs and overarching organizational goals.[1] The overall success of any software endeavor is heavily contingent upon the accuracy, detail, and completeness of these initial requirements.[3]

The importance of well-articulated requirements cannot be overstated. They form the primary basis for subsequent design, development, and testing activities. As highlighted in [1], "Software requirements can be thought of as the blueprint that guides the development and implementation of a software solution." Without such a blueprint, development efforts risk becoming misaligned, inefficient, and ultimately failing to deliver the intended value. An analogy often used to illustrate this point is that of constructing a building; [2] notes, "The more accurately you define the requirements, the less effort of your development team will be wasted, and the higher are the chances that you'll have the fully functional project delivered on time and with the lowest cost possible." This underscores the tangible benefits of meticulous requirements engineering in terms of resource optimization, adherence to timelines, and achievement of project outcomes. Furthermore, clearly defined requirements significantly reduce the risk of project delays and cost overruns by minimizing ambiguities that can lead to misunderstandings and scope creep—the uncontrolled expansion of project parameters.[1] They also enhance communication and foster collaboration among diverse stakeholders, including business users, IT teams, and external vendors, ensuring a shared understanding of the project's objectives.[1] This clarity also streamlines the process of selecting vendors, allowing for more objective comparisons of offerings based on their fit with established business needs.[1]

The consequences of inadequate requirements at the initial stages can create a cascading negative impact throughout the software development lifecycle. If requirements are vague, incomplete, or ambiguous [4], the design phase will inevitably be based on assumptions. Should these assumptions prove incorrect, they lead to flawed development. This, in turn, necessitates significant rework during the testing

phase or, in more detrimental cases, post-deployment. Such rework drastically increases project costs and extends timelines [2], and can also erode stakeholder confidence in the project and the development team.[5] Therefore, the initial investment in thorough requirements engineering is not merely a preliminary step but a critical risk mitigation strategy.

Beyond their technical function, requirements documents serve as a crucial communication contract among a diverse set of stakeholders, including business users, developers, testers, and clients. It is noted that requirements "facilitate effective communication between stakeholders" [1], and the Software Requirements Specification (SRS) often becomes the "sole source of truth" for the project.[6] This implies that the clarity, precision, and completeness of the requirements directly influence the shared understanding and alignment of all parties involved. Misinterpretations arising from poorly articulated requirements are a common and significant contributor to project failure.[4]

B. **An Overview of Requirement Categories and Their Hierarchy**

Requirements engineering is the comprehensive process of establishing the services that customers or users require from a system and the constraints under which that system must operate and be developed.[7] Within this discipline, software requirements are typically organized into several categories, often following a hierarchical structure. This hierarchy generally commences with high-level Business Requirements, which then translate into User Requirements. These, in turn, are further detailed into Product or Solution Requirements, a category that encompasses both Functional and Non-Functional Requirements.[3] System Requirements provide an even greater level of technical detail, specifying the characteristics of the software and hardware components.[10]

A clear articulation of this hierarchy is provided by sources such as [3], which states, "Business requirements come first in the gathering process. User requirement concentrates on objectives users can accomplish... Product requirements come last... non-functional and functional requirements... derive from the other requirements types." This establishes a distinct progression from broad strategic goals to specific system features. [8] further elaborates on this flow: Business requirements define high-level organizational objectives; Stakeholder (User) requirements express the needs of different user groups; and Solution requirements provide a detailed description of the system's expected features and characteristics, including both functional (what the system does) and non-functional (how well the system does it) aspects. Additionally, Transition Requirements address the specific needs for moving

from an existing state to the desired future state with the new system.[8]

The hierarchical nature of software requirements underscores the critical need for robust traceability. Traceability refers to the ability to follow the life of a requirement in both a forward and backward direction—from its origin through its development and specification, to its subsequent deployment and use, and through all periods of ongoing refinement and iteration in any of these phases.[4] For instance, if a business requirement dictates a 20% increase in market share, corresponding user requirements might specify features designed to attract new user segments. Functional requirements would then detail the specific behaviors of these features, and system requirements would outline the technical implementation. Without comprehensive traceability, it becomes exceedingly difficult to verify if a low-level technical decision or a specific system component still aligns with the overarching business objective. A change in a business requirement must be traceable down to all impacted system components and associated documentation, and conversely, the rationale for any system-level feature should be traceable back to a higher-level user or business need. The challenge of "Traceability and Impact Analysis" is a significant concern in requirements management.[4]

As one navigates down this hierarchy, from broad business objectives to detailed system specifications, the level of detail and technical specificity increases significantly. Business requirements are inherently high-level and strategic, often expressed in terms of market goals or operational efficiencies.[8] User requirements are typically articulated in natural language from the end-user's perspective, describing what they need to accomplish with the system.[14] Functional requirements then define the specific behaviors and actions the system must perform to meet these user needs.[16] Finally, system requirements provide a detailed technical blueprint, specifying architecture, components, and interfaces.[11] This progression reflects a crucial transformation from the "why" (business goals) and high-level "what" (user needs) to the detailed "how" (system implementation), ensuring that the final software product is a concrete and verifiable manifestation of initially abstract needs.

The following table summarizes the typical hierarchy of software requirements:

Requirement Type	Primary Focus	Level of Abstraction	Typical Artifacts/Examples	Key Sources

Business Requirements	High-level organizational goals, "Why"	Very High	Business case, project vision, market objectives	[3]
User Requirements	User needs and goals, "What" user wants to do	High	User stories, use cases, scenarios, natural language	[3]
Functional Requirements	Specific system behaviors/features, "What" does	Medium to Detailed	Feature list, function specifications, use case details	[3]
Non-Functional Reqs.	System quality attributes, "How well" does	Medium to Detailed	Performance targets, security policies, usability standards	[16]
System Requirements	Detailed technical specifications, "How" build	Very Detailed	Technical specs, interface definitions, architecture docs	[11]

This table visually and succinctly summarizes the distinct levels of software requirements, their primary purpose, and their progression from abstract business goals to concrete technical details. It helps in quickly grasping the relationships and flow within the requirements engineering process, which is fundamental to understanding the subsequent detailed sections on each requirement type.

II. User Requirements: The Voice of the Stakeholder

A. **Defining User Requirements: High-Level, Natural Language, Service-Oriented**

User requirements represent the needs and expectations of the end-users or other stakeholders who will interact with the software system. They are typically high-level, abstract statements that describe the services the system is expected to provide and the operational constraints under which it must function.[7] A defining characteristic of user requirements is their expression in natural language, often supplemented by

diagrams or tables. This approach ensures that they are readily understandable by end-users, customers, and other non-technical stakeholders who may not possess a deep technical background.[7]

[14] provides a concise and effective definition: "User requirements [are] High-level abstract requirements written as statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate." This captures the essence of their abstract nature and their focus on service provision. [15] reinforces this by stating that user requirements "outline the system's desired functionalities, features, and characteristics from the user's perspective." The "service-oriented" aspect is crucial; user requirements emphasize what the system *does for the user* or the tangible value it delivers to them or the business.[18] They articulate the goals users want to achieve with the system.

While the use of natural language makes user requirements accessible to a broad audience, it also inherently introduces the risk of ambiguity and misinterpretation. Natural language is described as "expressive, intuitive and universal" [7], but it is also noted that "requirements are expressed in the language of the application domain, which is not always understood by software engineers developing the system".[7] This potential for a "translation gap" means that a user's statement, such as "the system should be easy to use," can be interpreted in numerous ways by a development team. This necessitates further rigorous analysis and refinement of user requirements into more precise, verifiable functional and non-functional requirements. The challenge of "Ambiguous and Incomplete Requirements" is a significant issue in requirements management.[4]

Despite the potential for ambiguity, the natural language format of user requirements makes them critical for validation. Because they are expressed in terms understandable to the user, they serve as the primary instrument for confirming that the development team has correctly understood the user's needs *before* significant development effort is expended.[7][15] emphasizes "Validation and Feedback" as a key characteristic of the user requirements process. This early validation, based on user-centric language, helps prevent the development of a system that, while technically sound, fails to meet the actual needs of its users—a scenario that is far more costly and difficult to rectify later in the project lifecycle.

B. **Key Characteristics: User-Centricity, Stakeholder Involvement, Clarity**

Effective user requirements are fundamentally user-centric. This means they are defined and articulated from the perspective of the end-user, focusing on their goals,

tasks, and the value they expect to derive from the system.[15] Achieving this user-centricity necessitates deep involvement of users and other relevant stakeholders throughout the requirements elicitation, definition, and validation processes.[15] Furthermore, user requirements must be expressed in clear, concise, and unambiguous language, avoiding technical jargon that could obscure meaning for non-technical stakeholders.[15]

[15] explicitly highlights "User-Centric Approach," "Stakeholder Involvement," and "Clear and Concise Language" as pivotal aspects of well-defined user requirements. [15] elaborates on these, outlining practical best practices such as conducting thorough user research (e.g., surveys, interviews, observations), fostering close collaboration in workshops, and consistently avoiding overly technical terminology. The emphasis, as noted in [18]a, is on defining "'what' must be delivered to provide value to business when satisfied rather than 'how' to deliver." This focus on the desired outcome, rather than the implementation details, ensures clarity of purpose.

Active and continuous stakeholder involvement is particularly crucial for mitigating the problem of "implicitness" in requirements. Domain specialists or experienced users often have a deep understanding of their operational context and may take certain needs or constraints for granted, failing to articulate them explicitly.[7] Direct engagement techniques, such as interviews, focus groups, and workshops [19], as advocated by [15] and [15], compel these stakeholders to verbalize and document these implicit assumptions. This collaborative process leads to a more complete and accurate set of user requirements.

Clarity in user requirements also plays a vital role in enabling effective prioritization. When a user requirement clearly articulates the specific service it provides or the value it delivers to the user or business [18], it becomes significantly easier for stakeholders to assess its relative importance against other requirements and broader business objectives. [15] mentions "Prioritization and Trade-Offs" as an integral part of managing user requirements. Conversely, vague or poorly defined user requirements make it challenging to judge their necessity or potential impact, thereby hindering effective prioritization and resource allocation.

C. **Illustrative Examples of Service Provision**

User requirements should clearly describe the services or value the system is intended to deliver to its users or the business. These descriptions focus on the outcomes and benefits rather than technical implementation details.

[20] and [20] provide several excellent examples that illustrate this service-oriented nature:

- "Screen A accepts production information, including Lot, Product Number, and Date."
    - **Service/Value:** This requirement details a data capture service. The system provides value by enabling accurate and structured recording of critical production data, which is essential for manufacturing tracking, quality control, and inventory management.
- "System B produces the Lab Summary Report."
    - **Service/Value:** This describes an information generation service. The system delivers value by transforming raw data into a summarized report, which likely supports laboratory analysis, decision-making processes, or regulatory compliance reporting.
- "Twenty users can use System C concurrently without noticeable system delays."
    - **Service/Value:** This requirement highlights the provision of efficient concurrent access. The value lies in the system's ability to support multiple users simultaneously without performance degradation, which is crucial for productivity in collaborative environments.
- "System E will be compliant with 21 CFR 11."
    - **Service/Value:** This demonstrates a critical regulatory compliance service. For organizations in FDA-regulated industries, ensuring the system adheres to standards like 21 CFR Part 11 is paramount for data integrity, security, and avoiding legal penalties. The value is in risk mitigation and operational legitimacy.
- "Screen D can print on-screen data to the printer."
    - **Service/Value:** This illustrates a practical utility service, allowing users to create physical copies of digital information for record-keeping, distribution, or offline review.

These examples effectively show how user requirements, when well-defined, articulate the tangible benefits, functionalities, and services a system is expected to provide. It is also observable from these examples that user requirements often blend functional expectations with implicit non-functional qualities. For instance, the requirement "Twenty users can use System C concurrently without noticeable system delays" clearly describes a service (concurrent use) but also strongly implies a performance expectation ("without noticeable system delays"). This illustrates that while user requirements are primarily high-level descriptions of functionality, they often capture holistic needs that will later be decomposed into more discrete functional and non-functional system requirements during the refinement process.

III. Functional Requirements: Specifying System Behavior and Capabilities

A. **Core Definition and Purpose**

Functional requirements form the bedrock of a software system's specification, defining its core behaviors, features, and capabilities. They detail what the system must *do* in response to specific inputs or under particular conditions, and the outputs it should produce.[3] According to [16] and [16], "a functional requirement defines a function of a system or its component, where a function is described as a summary (or specification or statement) of behavior between inputs and outputs." Essentially, they specify the particular results a system is expected to deliver.[16]

[3] provides further clarity, stating that "functional requirements describe a system, its components, and the functions it must perform," characterizing them as "the system's basic behaviours." [21] echoes this, defining a functional requirement as "a statement of how a system must behave. It defines what the system should do in order to meet the user's needs or expectations." These requirements are pivotal as they directly drive the application architecture of the system, dictating how different software modules will be designed and interconnected to deliver the specified functionalities.[16]

Well-defined functional requirements serve as crucial anchors for system testability. Because they specify "particular results" [16] and should ideally be "measurable" [21], a corresponding test case can be designed for each functional requirement. This allows for verification that the system correctly performs the specified function when given certain inputs and produces the expected outputs. This direct linkage between functional requirements and testability is fundamental for effective quality assurance. If a function is not clearly and unambiguously defined, it cannot be effectively or reliably tested.

The common phrasing used for expressing functional requirements, such as "system must do <requirement>" [16], underscores their mandatory nature. For the system to fulfill its intended purpose and meet user needs, these functionalities must be present and operational. Unlike some non-functional requirements that might represent ideals or quality targets, functional requirements are typically binary in nature: either the system performs the specified function, or it does not. As noted in [55], "Functional requirements are mandatory for the system to work." If a functional requirement is not met, a piece of intended functionality is missing, which directly compromises the system's ability to satisfy user needs and business objectives.[8]

B. **Key Classifications with Detailed Examples:**

Functional requirements can be categorized based on the type of functionality they describe. The following classifications represent common areas of system behavior:

1. **Authentication**
    - **Definition:** Authentication is the process by which a system verifies the identity of a user, process, or device attempting to access its resources.[12] It answers the question, "Who are you?".
    - **Explanation and Examples:** A classic example of an authentication requirement is: "A user can register by providing a username, password, and email address. After registration, the user can log in using their username and password".[23] This can be more formally stated as: "The system must allow users to register with a unique username, a strong password, and a valid email address. The system shall verify the user's identity by comparing the provided username and password against stored credentials upon login".[23] Modern systems often include more sophisticated authentication methods, such as biometric verification (e.g., fingerprint or facial recognition) or multifactor authentication (MFA), which requires users to provide two or more verification factors.[12]
    - **Importance:** Authentication is a fundamental security measure. It ensures that only legitimate and verified entities can gain access to the system and its associated data, thereby preventing unauthorized access and potential misuse.
    - Authentication serves as a critical gateway to authorization. A system must first reliably establish *who* a user is through authentication before it can determine *what* that authenticated user is permitted to do, which is the domain of authorization.[23] This sequential dependency is paramount in security design; a compromised authentication mechanism effectively nullifies any subsequent authorization controls.
2. **Authorization**
    - **Definition:** Authorization is the process of determining whether an authenticated user, process, or device has the necessary permissions to access a specific resource or perform a particular action within the system.[8] It answers the question, "What are you allowed to do?".
    - **Explanation and Examples:** A clear example of an authorization requirement is: "Only users with an 'admin' level of access can view and modify system-wide configuration settings".[8] Another example: "The system shall restrict access to customer financial data to users belonging to the 'Finance Department' role".[856] notes that authorization functions "determine various system access levels and decide who can CRUD (change, read, update, or

delete) information."

- **Importance:** Authorization enforces access control policies, such as Role-Based Access Control (RBAC) or Access Control Lists (ACLs). It is essential for protecting sensitive data, ensuring data integrity, and limiting users' interactions to only those parts of the system and functionalities that are relevant to their roles and responsibilities.
- The granularity of authorization rules significantly impacts both system complexity and security. A system with very fine-grained authorization (e.g., permissions at the level of individual data fields or specific operations on objects) can offer very precise security control. However, it also increases the complexity of managing these permissions and the underlying authorization logic. Conversely, coarser-grained authorization (e.g., simple user vs. admin roles) is simpler to manage but may offer less nuanced security. This trade-off between security precision and administrative complexity is a key consideration when defining authorization requirements.

3. **Data Parsing**
   - **Definition:** Data parsing involves the analysis of a string of symbols, special characters, or data structures, typically to convert data from one format to another. This often means transforming unstructured or semi-structured data into a structured, machine-readable format that can be easily processed by software applications.[25]
   - **Explanation and Examples:** An example is converting a PDF invoice into a structured JSON object containing fields like invoice_number, vendor_name, item_details, and total_amount.[25] A functional requirement could be: "The system shall parse incoming comma-separated value (CSV) files containing product information and extract product_id, product_name, price, and stock_quantity for import into the product database".[25] The process of data parsing often involves steps like data acquisition, preprocessing (cleaning and standardizing), the parsing itself (which might use techniques like text analysis, structured data extraction, or pattern recognition), post-processing (validation and refinement), and finally, integration into target systems.[25]
   - **Importance:** Data parsing is crucial for enabling data exchange between disparate systems, facilitating system interoperability, supporting data migration, preparing data for analysis and reporting, and automating the processing of diverse data sources like logs, user inputs, or external data feeds.
   - The increasing volume and complexity of modern data, particularly unstructured data from sources like text documents, images, and social media, is driving a significant trend towards the use of Artificial Intelligence

(AI) and Machine Learning (ML) techniques for more robust and intelligent parsing. [25] refers to "AI-driven data parsing" and distinguishes between "grammar-driven" approaches and "data-driven data parsing" that utilizes "smart statistical parsers and modern treebanks." [25] also emphasizes that "AI models analyze the data's structure." This shift from purely rule-based parsers to more adaptive, learning-based systems capable of handling ambiguity, variations, and novel data formats is a notable development. Consequently, functional requirements for data parsing might now also need to specify aspects like expected accuracy rates or the system's learning capabilities if AI/ML components are involved.

4. **UI/UX Functional Aspects**
    - **Definition:** These are functional requirements that specify the interactive elements, navigational pathways, and behavioral responses of the user interface (UI) necessary to enable users to effectively interact with the system and achieve their goals. While User Experience (UX) is a broader quality attribute often considered non-functional, certain UI functionalities are prerequisites for a usable system.
    - **Explanation and Examples:** [22] states, "UI requirements specify how your users will interact with your product. They define design elements that make navigation intuitive." An example provided is: "The dashboard must display account balance, available buying power, and top stock movers of the day." This is functional because it dictates *what* specific information must be presented on the dashboard. [57] provides further UX-focused functional examples: "Users must be able to navigate through a product catalog and filter products by categories," and "Implementation of a search bar that allows users to search for information within the site using keywords." Other examples include: "The system shall provide a 'Reset Password' button on the login screen," or "Upon successful form submission, the system shall display a confirmation message to the user."
    - **Importance:** These requirements ensure that the system provides the necessary interactive capabilities and information displays for users to perform their tasks efficiently and effectively. They form the tangible points of interaction between the user and the system's underlying logic.
    - It is important to distinguish between functional UI requirements and non-functional usability requirements. Functional UI requirements define the presence and basic behavior of UI elements (e.g., "A 'Submit' button shall exist on the data entry form"). Non-functional usability requirements, on the other hand, define the quality of that interaction (e.g., "The 'Submit' button must be clearly visible and easily discoverable," or "The form submission

process must provide feedback to the user within 1 second"). [22] differentiates functional (what the system does) from non-functional (how it does it). [57] explicitly separates Functional Requirements in UX (e.g., "search bar implementation") from Non-Functional Requirements in UX (e.g., "pages must load in 2 seconds," "intuitive interface"). A system can possess all the necessary functional UI elements but still offer a poor user experience if related non-functional requirements like performance, intuitiveness, and accessibility are neglected.

5. **Reporting**
    - **Definition:** Reporting functional requirements specify the system's ability to generate, display, format, and distribute information in a structured and meaningful way to meet user, business, or regulatory needs.
    - **Explanation and Examples:** [13] provides an example: "Reporting – Generates sales reports and data processing for revenue data reports." More detailed examples from [27] include: "The system must allow users to create and customize reports based on specific criteria and filters to meet diverse informational needs," "The system must include tools for visualizing data through charts, graphs, and other graphical elements to enhance data interpretation and presentation," and "The system must offer data aggregation and drill-down functionalities, allowing users to summarize data at higher levels and explore detailed data at granular levels as needed." [28] and [28] offer: "The system will generate standard payroll reports viewable in PDF format. Users can customize report parameters like date range and employee filters."
    - **Importance:** Reporting capabilities are vital for providing insights into system operations or business performance, supporting informed decision-making, tracking key metrics, fulfilling compliance obligations, and facilitating operational oversight.
    - Effective reporting functionality serves as a bridge, transforming raw data into actionable intelligence. Reports are not merely about displaying data [13]; they are fundamentally about enabling users to "meet diverse informational needs" [27], which implies facilitating analysis and supporting decision-making processes. The functional capabilities to customize parameters, apply filters [27], visualize data through charts and graphs, and perform drill-downs [27] directly enhance the value of the information presented. These features empower users to turn vast amounts of data into a strategic asset. The complexity of reporting requirements can vary significantly, from simple tabular data dumps to sophisticated, interactive business intelligence dashboards.

6. **System Integration**
    - **Definition:** System integration functional requirements define how the software system will interact, exchange data, or invoke services with other software systems, hardware components, or third-party services, whether they are internal or external to the organization.[12]
    - **Explanation and Examples:** [22] states, "Integration requirements define how your product will interact with other systems or services." An example given is: "The app must connect to financial data feeds in real-time to display stock prices and market trends." The integration requirements tab in [27] details specific interactions, including data formats and frequency of exchange, such as integrating "Student information (e.g., Name, PUID, Major, etc.)" from a "Campus Solutions" system via a "csv file" on a "Once a day" basis.[2712] notes that these requirements describe "how the system interacts and integrates with other systems or third-party services."
    - **Importance:** System integration is essential for creating larger, more capable enterprise ecosystems, leveraging existing functionalities from other systems, ensuring seamless data flow across different business units or applications, and enabling interaction with external partners or services.
    - Application Programming Interfaces (APIs) have become the linchpin of modern system integration, and their functional requirements are critical for successful interoperation. [27] mentions "API call" as a type of integration. [47] extensively discusses APIs as a primary type of external interface, emphasizing the need to meticulously define "input parameters, specifying expected output formats, outlining authentication mechanisms, and establishing data exchange protocols." The functional requirements for an API—detailing the operations it supports, the data structures it accepts as input, and the data it returns as output—are fundamental to its usability by other systems and the overall success of the integration effort. Poorly defined or incomplete API functional requirements are a common cause of integration failures and complexities.

The following table provides a comparative overview of functional and non-functional requirements:

Feature	Functional Requirements	Non-Functional Requirements (NFRs)	Key Sources

Focus	What the system *does*	How well the system *performs* or its qualities/constraints	[16]
Purpose	Define specific behaviors, features, system services	Define quality attributes, operational constraints, user experience	[16]
Nature	Typically mandatory for system operation	Often define targets, levels, or constraints on functionality	[43]
Expression	"System shall do X," "System must provide Y"	"System shall be Z," "System must achieve P performance"	[16]
Testability	Directly testable (input -> output verification)	Often indirectly testable, may require specific environments/tools	[29]
Examples	User login, data calculation, report generation	Performance (response time), security (encryption), usability	[16]
Impact if Not Met	System fails to perform intended tasks or provide features	Poor user experience, system instability, security breaches	[43]
Driver For	Application Architecture	Technical Architecture	[16]

This table is crucial as it directly addresses a core distinction in requirements engineering, providing a quick, side-by-side comparison of the fundamental differences between what a system does and how well it does it. This clarity helps in understanding how both types of requirements contribute to the overall success of a software product and aids in their proper elicitation and specification.

IV. Non-Functional Requirements: Defining System Quality and Constraints

A. **Core Definition and Significance**

Non-Functional Requirements (NFRs) are critical specifications that define the operational attributes, quality characteristics, and constraints of a software system, rather than its specific behaviors or functions.[16] While functional requirements describe *what* a system should do, NFRs describe *how well* the system performs those functions or the conditions under which it operates.[29] They act as constraints on the design or implementation choices, covering aspects such as performance, security, reliability, and usability.[16]

[29] provides a clear definition: "non-functional requirements, also known as quality attributes, are specifications that define the operational attributes of a system rather than its specific behaviors." Similarly, [30] states that "NFRs address how the system behaves under various conditions rather than what the system does," and they "describe how well a system functions." These requirements are fundamental because they drive the technical architecture of the system [16], influencing choices about hardware, software platforms, and design patterns.

The significance of NFRs cannot be understated. They are crucial for ensuring a positive user experience, maintaining system reliability and stability, and achieving overall system effectiveness.[31] As noted in [43], insufficient attention to NFRs can lead to poor user experiences, even if all functional requirements are met. A system might perform all its intended tasks correctly but be so slow, insecure, or difficult to use that it fails to meet stakeholder expectations or business objectives.

Meeting NFRs, particularly those that are stringent (e.g., very high availability or extremely fast response times), often significantly increases the development cost and effort. [29] explicitly states, "Non-functional requirements generally increase the cost as they require special efforts during the implementation." Furthermore, [43] observes that "Excess non-functional requirements can quickly drive up the cost." For example, achieving 99.999% availability or sub-second response times under heavy concurrent load typically requires sophisticated architectural designs, extensive redundancy in hardware and software, and specialized technologies, all of which contribute to higher expenses. This makes the careful definition, negotiation, and prioritization of NFRs a critical activity from both a technical and a budget perspective.

Another important consideration is that NFRs are often interdependent and can

sometimes conflict with each other, necessitating careful trade-offs during the design process. [29] highlights that "Non-functional requirements may be in conflict with each other and it can be difficult to balance them." For instance, implementing robust security measures (e.g., complex data encryption algorithms, multiple authentication steps) might introduce latency and thus negatively impact system performance. Similarly, designing for high scalability might involve architectural choices that increase the complexity of system maintenance. These interdependencies mean that NFRs cannot be defined or addressed in isolation; they must be considered as a holistic set of quality goals. Trade-offs must be explicitly discussed, documented, and agreed upon with stakeholders to ensure a balanced and achievable set of system qualities.

B. **Key Classifications with Detailed Examples:**

NFRs are typically categorized based on the quality attribute they address. The following are key classifications:

1. **Scalability**
    - **Definition:** Scalability refers to the system's ability to efficiently handle an increasing amount of work—such as a growing number of users, larger data volumes, or higher transaction rates—by adding resources (e.g., servers, processing power, memory) while maintaining acceptable performance levels.[12]
    - **Explanation and Examples:** [31] and [31] define scalability NFRs as those that "measure the system's ability to maintain performance levels as the workload increases." A common example is: "An e-commerce website should maintain its average page load time of under 3 seconds when the number of concurrent users increases from 1,000 to 10,000 during a peak sales event".[3133] provides another: "The system should be capable of scaling from 1,000 to 10,000 active users with minimal impact on performance (e.g., less than 10% degradation in response time) by adding additional server resources without requiring significant changes to the system architecture." [27] suggests phrasing like: "System should be designed to support a 50% increase in user load and data volume over the next 24 months, allowing for seamless scaling of resources."
    - **Importance:** Scalability is crucial for ensuring that the system can grow in alignment with business needs and can handle anticipated peak loads without performance degradation, service interruptions, or the need for costly emergency re-engineering.
    - Designing for scalability from the outset is far more effective and less costly

than attempting to retrofit scalability into a system not initially built for growth. The statement in [33] about adding resources "without significant changes to the system architecture" implies that the architecture itself must be inherently designed to accommodate scaling (e.g., through modularity, stateless services, or distributed data management). Attempting to scale a monolithic architecture not designed for it often leads to major re-engineering efforts and significant downtime. Therefore, scalability requirements heavily influence early architectural decisions.[16]

2. **Performance**
    - **Definition:** Performance NFRs specify how efficiently and quickly the system responds to user actions or processes tasks under defined conditions. Key aspects include response time (latency), throughput (e.g., transactions per second), and resource utilization (e.g., CPU, memory, network bandwidth).[12]
    - **Explanation and Examples:** [34] and [34] define performance NFRs covering these aspects. Examples include: "The website pages must load completely within 3 seconds for 95% of users accessing via a broadband connection, with up to 5,000 concurrent visitors".[35] "A search engine must return query results within 1 second for typical queries".[3127] adds: "System should maintain optimal performance levels, including handling peak usage times without degradation, ensuring quick response times, and meeting predefined performance benchmarks under various load conditions."
    - **Importance:** Performance directly and significantly impacts user satisfaction, productivity, and the overall perception of system quality. Slow or unresponsive systems lead to user frustration, task abandonment, and can negatively affect business outcomes.[34]
    - Vague performance goals are of little practical use. To be effective, NFRs for performance must be specific, measurable, and verifiable against defined benchmarks or load conditions. [29] emphasizes that NFRs should be "quantitative, measurable and testable." [34] highlights the importance of "Performance Testing (load testing, stress testing)" for validation. A statement like "the system should be fast" is insufficient. A well-defined performance NFR, such as "95% of search queries must return results in under 2 seconds when the system is under a load of 1000 concurrent users" [31], provides a clear, testable target for developers and quality assurance teams.
3. **Portability**
    - **Definition:** Portability refers to the ease with which a software system or its components can be transferred from one hardware or software environment (e.g., operating system, database, browser) to another, with minimal or no modification.[12]

- **Explanation and Examples:** [36] and [36] state, "Portability requirements ensure the system works well on different platforms and in different environments." An example given is: "The fitness application must operate smoothly and consistently on both Android (version 8.0 and above) and iOS (version 13.0 and above) devices, syncing user data seamlessly across all logged-in platforms." [33] provides further examples: "Cross-Platform Support: The application must be operable on both Windows 10/11 and macOS (latest two versions) without requiring any code modifications." and "Browser Compatibility: The web application should be fully compatible with the latest stable versions of Chrome, Firefox, Safari, and Edge browsers."
- **Importance:** Portability allows software to reach a wider user base, adapt to evolving technology landscapes, reduce dependency on specific vendors (vendor lock-in), and potentially lower long-term maintenance costs if migration to new platforms becomes necessary.
- Achieving a high degree of portability often requires careful and early selection of technologies, such as cross-platform development frameworks or languages known for broad compatibility (e.g., Java with its Java Virtual Machine). It may also involve the use of abstraction layers within the software architecture to isolate system-dependent components. For instance, to ensure an application runs on both Windows and macOS [33], developers might opt for a language like Python with appropriate libraries or a framework like Electron. These portability NFRs, when defined early in the project lifecycle, can significantly constrain architectural choices and the technology stack. If such requirements are not considered upfront, a system might become deeply intertwined with a specific platform, making future porting efforts extremely costly, time-consuming, or even practically infeasible.

4. **Reliability**
   - **Definition:** Reliability is the ability of a software system to perform its required functions consistently and correctly under stated conditions for a specified period. It is often measured by metrics such as Mean Time Between Failures (MTBF) or the probability of failure-free operation.[12]
   - **Explanation and Examples:** [37] and [37] note that reliability NFRs specify "how often the system will fail" and include factors like Mean Time To Failure (MTTF) and Mean Time To Repair (MTTR). An example often linked to reliability (though also to availability) is: "The system must achieve an operational availability of 99.9% during business hours".[3732] elaborates that reliability "defines the ability of the system to properly perform the required functions under predefined conditions for a certain period of time. Commonly, it's expressed through probability percentages predicting chances that the

system won't experience critical failure..." An example focusing on data integrity aspect of reliability from [12] is: "A database update process must roll back all related updates atomically if any single update within the transaction fails."

- **Importance:** Reliability is crucial for maintaining user trust, ensuring business continuity, and is paramount in systems where failure can lead to significant financial loss, data corruption, or safety hazards (e.g., financial trading systems, medical devices, industrial control systems).
- Highly reliable systems are generally not those that are naively expected to never fail, but rather those that are designed to anticipate potential failures and are engineered to handle them gracefully. This often involves incorporating mechanisms for fault tolerance, redundancy of critical components, robust error detection and handling, and effective recovery procedures.[32] The example from [12] regarding the atomic rollback of failed database updates illustrates a design strategy to maintain data consistency (an aspect of reliability) despite partial failures. This implies that reliability requirements drive the need for specific architectural patterns and sophisticated error-handling strategies that extend beyond simply writing correct code for the "happy path" scenarios.

5. **Maintainability**
   - **Definition:** Maintainability refers to the ease and efficiency with which a software system can be modified to correct faults (corrective maintenance), improve performance or other attributes (perfective maintenance), or adapt to a changed environment (adaptive maintenance).[12]
   - **Explanation and Examples:** [38] and [38] define maintainability as "how easy it is for a system to be supported, changed, enhanced, and restructured over time." Factors influencing it include the expected lifespan of the software, the anticipated frequency of revisions, and the availability of skilled support resources. An example from [31] is: "A web application should allow developers to resolve critical bugs (Severity 1) within 2 business hours of verified detection." [33] offers more structural examples: "Modifiability: Any changes or updates to individual system modules should not unintentionally affect more than 5% of the existing codebase in other modules," and "Documentation: All code modules and public APIs should be documented to at least 90% completeness according to the project's defined documentation standards."
   - **Importance:** Maintainability significantly affects the total cost of ownership (TCO) over the software's entire lifecycle. Systems with poor maintainability are difficult and expensive to update, fix, or enhance, leading to increased operational costs and slower adaptation to new business requirements.

- While users do not directly "see" or interact with maintainability as a feature, they experience its effects indirectly through faster resolution of bugs, quicker availability of new features and enhancements, and overall system stability and adaptability over time. [38] highlights that improperly assessing and addressing maintainability can lead to "serious issues further down the line." This is because a system that is difficult to maintain becomes resistant to change. As business needs inevitably evolve, the inability to adapt the software quickly and cost-effectively (due to poor maintainability) can render the system obsolete or a significant operational liability. Therefore, investing in practices that promote maintainability—such as good architectural design, clean and modular code, comprehensive documentation [33], and automated testing—is an investment in the system's long-term viability and value.

6. **Availability**
   - **Definition:** Availability is the degree to which a system, subsystem, or component is operational and accessible when required for use by authorized users. It is commonly expressed as a percentage of uptime over a given period.[12]
   - **Explanation and Examples:** [39] and [39] define high availability as "the ability of a system to operate continuously, without failure, for a significantly long period." Availability is typically measured using the formula: Availability (%) = (Total operational time / Total time) x 100. The concept of "nines" is often used, where "five nines" (99.999%) availability translates to approximately only 5 minutes of downtime per year.[39] An example from [31] is: "A customer support portal should aim for less than 1 hour of unplanned downtime per month during standard business hours (9 AM - 5 PM, Monday-Friday)." [27] suggests defining "acceptable downtime windows, such as weekends from 18:00 to 20:00, to perform necessary maintenance and updates with minimal impact on users."
   - **Importance:** Availability is critical for business operations, customer satisfaction, and revenue generation, especially for online services, e-commerce platforms, and mission-critical enterprise systems where downtime can lead to immediate and significant negative consequences.
   - Since achieving 100% availability is practically impossible and prohibitively expensive [39], strategies for high availability focus on minimizing the frequency and duration of downtime when failures inevitably occur. [40] discusses that striving for 100% availability "is a potentially expensive option as it means developing solutions to ensure that in the event of component(s) failure, the solution can compensate (for example a full 'mirror' of the solution is always running in parallel)." [33] also mentions "automatic failover to a secondary data

center in case the primary data center goes down." This demonstrates that high availability is not solely about preventing individual component failures (though component reliability contributes significantly to it). More importantly, it is about system-level architectural design for resilience, which includes redundancy of hardware, software, and data, along with automated failover mechanisms to ensure service continuity or rapid recovery.

The following table summarizes key NFR classifications with example metrics to enhance their measurability:

NFR Category	Brief Description	Quantifiable Example Metric	Key Sources
Scalability	System's ability to handle increased load by adding resources.	"System must support 10,000 concurrent users with <10% increase in average response time per doubling of users."	27
Performance	System's responsiveness, throughput, and resource utilization.	"95% of API responses shall be delivered in < 200ms under a sustained load of 500 requests per second."	27
Portability	Ease of transferring system to different environments.	"Application must be fully functional on Windows 10/11 and macOS Monterey/Ventura with no code changes required."	33
Reliability	System's ability to operate without failure for a specified time/conditions.	"Mean Time Between Failures (MTBF) for critical financial transactions shall be greater than 1000 operational hours."	31

Maintainability	Ease of modifying, correcting, or enhancing the system.	"Average time to diagnose and fix critical defects (Severity 1) shall be less than 4 business hours."	[31]
Availability	Percentage of time the system is operational and accessible.	"System shall achieve 99.99% uptime (max 52.56 minutes of downtime per year), excluding scheduled maintenance windows (max 2 hours/month)."	[27]

This table is valuable because NFRs can often seem abstract. By pairing each NFR category with a concrete, quantifiable example metric, it makes them more understandable, measurable, and testable, supporting the need for NFRs to be verifiable.[7]

V. System Requirements: The Detailed Technical Blueprint

A. **Defining System Requirements and Their Role**

System requirements are detailed descriptions of the services the system must provide and the constraints under which it must operate. They are typically generated during the requirements engineering process and represent a transformation of the stakeholder's view of desired capabilities into a technical, developer-oriented perspective of how the system can achieve those capabilities.[7] These requirements specify the technical characteristics of the software system, including its architecture, necessary hardware and software components, and the interfaces it will have with other systems or users.[17]

[11] provides a comprehensive definition: "System requirements describe requirements which the system-of-interest (SoI) must fulfill to satisfy the stakeholder needs and are expressed in an appropriate combination of well-formed textual statements and supporting models or diagrams." Crucially, these system requirements form the direct basis for subsequent system architecture, detailed design, system integration activities, and verification processes.[1117] further clarifies that "System requirements... are typically expressed in technical terms and are often used as a basis for system design." While user requirements are high-level and abstract, system requirements

delve into greater detail; for instance, "Functional system requirements should describe the system services in detail".[7]

System requirements serve as a critical bridge between the "what" (defined by user and functional requirements) and the "how" (detailed in design and implementation). [11] notes that system requirements transform the "stakeholder view... into a technical, developer view." [7] observes the close relationship between requirements and design, stating that they are often "inseparable." System requirements occupy this crucial juncture, providing sufficient technical detail to guide the design process effectively without over-constraining innovative solutions. They represent the point where abstract user needs and functional goals begin to meet concrete technical possibilities and constraints.

The creation of system requirements is not a standalone activity but rather the result of a careful derivation and refinement process originating from higher-level user and business requirements. As explained in [11], the "System Requirements Definition uses the outcome of the System Concept Definition activities," which are themselves based on elicited stakeholder needs. The process involves "transforming Needs to System Requirements." This iterative refinement is essential to ensure that the evolving technical solution remains firmly aligned with the original goals and intended value proposition. A failure in this careful transformation can lead to the development of a technically sound system that, unfortunately, fails to meet the actual needs of its users or the business.

B. **Essential Subclasses:**

System requirements can be further categorized into several subclasses, each addressing specific aspects of the technical specification:

1. **Functional System Requirements (Detailed System Services & Operations)**
    - **Definition:** These are detailed specifications of the system's behaviors, operations, and services, elaborating on the higher-level functional requirements from a comprehensive system perspective.[7] They describe precisely what the system will do.
    - **Explanation and Examples:** [7] distinguishes these from functional user requirements, stating, "Functional system requirements should describe the system services in detail," whereas "Functional user requirements may be high-level statements." While [12] (Altexsoft) discusses general functional requirements, many examples provided are detailed enough to be considered functional system requirements. For instance, "The system shall send a confirmation email to the user upon new user account creation, containing a

unique verification link that expires within 24 hours." This specifies an input (account creation), a detailed process (email generation, unique link creation with an expiry), and an output (email sent). These requirements articulate inputs, processing logic, outputs, and interactions with data stores and other system components at a granular, unambiguous level.

- **Importance:** Functional system requirements provide the precise, unambiguous specifications that developers need to correctly implement the system's functionalities. They are the direct input to the design and coding phases. Abstract user requirements or even general functional requirements might still lack the specificity needed for implementation. Functional *system* requirements, however, must be sufficiently detailed [7] to allow a designer to map them to specific system components (e.g., modules, classes, functions) and for a programmer to implement the underlying logic. For example, a general functional requirement like "The system shall calculate sales tax" would be refined into a functional system requirement such as: "The system shall, upon receiving an order with a shipping address, retrieve the applicable state and local sales tax rates from the Tax Rate Service based on the ZIP code of the shipping address. It shall then calculate the sales tax by multiplying the pre-tax order total by the combined applicable tax rate and round the result to two decimal places. This calculated sales tax amount shall be stored with the order details and displayed to the user on the order confirmation screen."

2. **Non-Functional System Requirements (System-Level Quality Attributes & Constraints)**
   - **Definition:** These are detailed specifications of the quality attributes (e.g., performance, security, reliability) and operational constraints that apply to the system as a whole or to its major components. They are derived and quantified from general NFRs identified earlier in the requirements process.[12]
   - **Explanation and Examples:** [12] (Altexsoft) notes that "Nonfunctional requirements describe the general properties of a system. They are also known as quality attributes." The examples provided, such as "Website pages must load within 3 seconds with up to 5,000 simultaneous users," are specific enough to be considered system-level NFRs. [42] defines NFRs as outlining "a system's quality attributes, performance standards, and operational constraints," citing "The system must process transactions within two seconds" as an example. These requirements translate broader NFR aspirations (like "the system should have high performance") into specific, measurable, achievable, relevant, and time-bound (SMART) targets for the system being built. For example, a general security NFR of "the system must

be secure" might be refined into non-functional system requirements like: "All sensitive user data (as defined in Appendix X) stored in the database must be encrypted using AES-256 encryption," and "The system must withstand common web application vulnerabilities as per the OWASP Top 10, verified by penetration testing."

- **Importance:** Non-functional system requirements are crucial for ensuring that the system meets overall quality standards. They heavily influence architectural and design decisions, as achieving specific system-level NFRs often necessitates particular architectural patterns or technology choices. For instance, a system NFR stipulating 99.999% availability or sub-second response times for one million concurrent users would likely compel architects to consider a distributed microservices architecture, advanced load balancing, specific database technologies optimized for performance and scalability, and robust caching strategies. These are fundamental design decisions driven directly by system-level NFRs.[16]

3. **Technical Specifications (The "How-To" Implementation Guide)**
   - **Definition:** Technical specifications are comprehensive documents that provide an in-depth outline of the detailed technical requirements, overall design, and specific functionality of a product, system, or project. They serve as a vital blueprint for engineering and development teams, effectively bridging the gap between high-level business requirements and the concrete technical implementation.[44]
   - **Explanation and Examples:** According to [44] and [44], "A technical specification... is a comprehensive document outlining the detailed requirements, design, and functionality... It covers all the vital, nitty-gritty information about the product development process and lays out the technical requirements necessary to achieve the project's goals." It typically focuses on aspects like "internal programming, technical standards, and performance requirements." [44] draws a key distinction: "A functional specification describes what you want... a technical specification details how you get there." A technical specification document typically includes sections on functional requirements, design requirements (including UI/UX design elements), applicable technical standards and protocols, the proposed technical solution and approach for each project element, testing requirements (including test plans and acceptance criteria), delivery requirements (installation, deployment), support and maintenance plans, criteria for success evaluation, and a detailed work breakdown structure with timelines.[44]
   - **Importance:** Technical specifications are paramount for providing clarity and

precision, thereby reducing ambiguity and the risk of misunderstandings. They help in identifying potential technical challenges early, improving the quality of the final product, boosting development efficiency by minimizing errors and rework, and ensuring that the technical implementation aligns with overarching business goals.[44]

- While comprehensive, upfront technical specification documents are traditionally associated with waterfall development methodologies, the *need* for specifying technical details persists even in agile environments. [45] mentions that a tech spec helps the Project Manager plan work and Quality Assurance engineers to test effectively. In agile contexts, this might translate to more granular technical specifications embedded within user stories or detailed task breakdowns created just-in-time before an iteration or sprint. The concept of the SRS as a "living document" [6], while referring to the broader requirements specification, applies conceptually here too; technical details evolve. The core purpose—providing unambiguous technical clarity for implementation—remains essential, regardless of the development methodology, as the risk of "misunderstandings and development errors" [44] is universal.

4. **Interface Requirements (Defining Interactions with External Entities)**
   - **Definition:** Interface requirements are specifications that define how the software system will interact and exchange information or services with other systems, software components, hardware devices, or human users. These requirements detail the data formats, communication protocols, methods of interaction, and any necessary security measures for these interfaces.[11]
   - **Explanation and Examples:** [46] and [46] state that "Interface requirements refer to the specifications or conditions that define how different systems, software, or components will interact with one another... [they] describe the methods, data formats, and protocols used..." [47] and [47] categorize types of external interfaces, including:
     - **Application Programming Interfaces (APIs):** Defining input parameters, expected output formats, authentication mechanisms, and data exchange protocols (e.g., "The Order Management System shall expose a REST API endpoint /orders/{orderId} that accepts a GET request and returns the order details in JSON format.").
     - **Web Services:** Specifying message formats (e.g., XML, JSON) and communication protocols (e.g., SOAP, REST) for internet-based system-to-system communication.
     - **Data Integration Interfaces:** Outlining data formats, transfer mechanisms (e.g., batch file transfer, real-time streaming via message

queues), and synchronization rules for data exchange between databases or systems.
    - **Hardware Interfaces:** Specifying communication protocols (e.g., USB, Modbus), command formats, and data exchange mechanisms for interaction with physical devices like sensors, printers, or industrial controllers. [11] also mentions "interface analysis" and the need for "defined interactions across boundaries" as integral parts of the system requirements definition process.
  - **Importance:** Interface requirements are essential for ensuring interoperability, allowing the system to function effectively and seamlessly within its broader operational environment, which often includes a multitude of other interconnected systems. A lack of clear and precise interface requirements is a common source of integration problems, delays, and increased development costs.[46]
  - Well-defined interface requirements effectively act as formal "contracts" between the interacting components or systems. When System A needs to communicate with System B, the interface specification [47] dictates the precise rules of engagement. Both systems must strictly adhere to this "contract" for successful interaction. Any unilateral deviation by one system can break the communication flow and lead to integration failures. This contractual nature is especially critical for external interfaces [47], where the development teams for the interacting systems might belong to different organizations or vendors. Clear, unambiguous interface requirements significantly reduce the likelihood of "he said, she said" disputes and finger-pointing during the integration and testing phases. [46] even provides an example of how an interface requirement might be formally stipulated in a legal agreement between parties.

The following table helps to distinguish the primary focus and scope of these system requirement subclasses:

Subclass	Primary Focus/Purpose	Level of Detail	Relationship to Other Req. Types	Key Sources
Functional System Req.	Detailed specification of *what* services the system	Very Detailed	Elaboration of User & general Functional Req. for technical	[7]

	provides.		implementation.	
Non-Functional System Req.	Detailed specification of *how well* system performs (quality).	Very Detailed	Concrete targets for general NFRs at the system level.	[12]
Technical Specifications	The overall "how-to" blueprint for building the system.	Comprehensive	Encompasses functional/NFRs, design, standards, implementation plan.	[44]
Interface Requirements	How the system interacts with external/internal entities.	Detailed	Specifies protocols, data formats for system integrations (functional).	[11]

This table aids in differentiating the specific roles and scopes of various system requirement subclasses. These terms can sometimes be used interchangeably or overlap in practice, so a clear distinction of their primary focus, detail level, and relationship to other requirement types provides essential clarity for understanding the full spectrum of technical documentation needed for successful software development.

VI. Best Practices in Requirements Documentation and Management

A. **The Software Requirements Specification (SRS) Document**

The Software Requirements Specification (SRS) document is a cornerstone of effective software development. It is a comprehensive document that meticulously describes the complete behavior, features, and constraints required of the software *before* it is designed, built, and tested.[48] An SRS typically includes a detailed account of both functional and non-functional requirements, specifications for external interfaces (how the system interacts with users, hardware, and other software), and any applicable design or implementation constraints.[3]

According to [5] and [5], the SRS (also referred to as a Software Requirements Document or SRD) "includes guidelines for how the developers should develop the software...

and the client/ investor expectations regarding the purpose and outcomes of the finished product." It often incorporates use cases to illustrate user interactions and system responses. [6] and [6] further emphasize that an SRS lists "requirements, expectations, design, and standards," encompassing high-level business requirements, specific end-user needs, and the product's functionality described in technical terms. Crucially, the SRS often becomes the "sole source of truth" for the project, ensuring all stakeholders and team members have a consistent and agreed-upon understanding of what is to be built.[6]

The benefits of a well-crafted SRS are manifold. It boosts stakeholder confidence by providing a clear articulation of the intended product, allows for unambiguous communication and alignment across teams, offers a detailed blueprint for the development process, leads to a more structured and predictable development lifecycle, and serves as an essential foundation for subsequent testing and validation activities.[5] Furthermore, a good SRS can significantly reduce overall development effort by catching misunderstandings early, provides a solid basis for realistic cost and schedule estimates, and facilitates verification and validation.[48]

The key components of a comprehensive SRS, as synthesized from sources like [5], and [6], generally include:

- **Introduction:** This section sets the stage, outlining the Purpose of the SRS, its intended Scope, Definitions of terms and acronyms used, a list of References to other relevant documents, and an Overview of the SRS document structure.
- **Overall Description (or General Description):** This part provides context, describing the Product Perspective (how it fits with other systems or business processes), a summary of Product Functions (often with use case diagrams), characteristics of the intended User Classes, general Constraints (e.g., regulatory, hardware), and any Assumptions and Dependencies.
- **Specific Requirements:** This is the core of the SRS, detailing all requirements with sufficient precision. It typically covers:
    - Functional Requirements (specific behaviors and operations)
    - Non-Functional Requirements (quality attributes like performance, security, usability)
    - External Interface Requirements (interactions with users, hardware, other software, and communication systems)
    - Logical Database Requirements (if applicable)
    - Design Constraints (e.g., standards to be followed, specific technologies to be used)
    - Software System Attributes (e.g., reliability, availability, maintainability,

portability)
- **Appendices and Index (if necessary):** For supplementary information, glossaries, or detailed models.

Adherence to established standards, such as the IEEE 830 "Recommended Practice for Software Requirements Specifications," can greatly enhance the quality and utility of an SRS.[48] IEEE 830 provides guidance on the content, qualities of a good SRS, and presents several sample outlines. The benefits cited for using IEEE 830 align with the general advantages of a good SRS, including establishing a clear basis for agreement between customers and suppliers, reducing development effort through early clarification, providing a foundation for realistic estimates and effective validation and verification, facilitating the transfer of the software product to new users or platforms, and serving as a baseline for future enhancements.[48] The typical structure outlined in IEEE 830 (Introduction, General Description, Specific Requirements, and Additional Information) is consistent with best practices for SRS documentation.[48]

While the SRS has traditionally been viewed as a comprehensive, fixed document created before the design phase (particularly in waterfall methodologies), its role and format must adapt in the context of agile and iterative development. [6] aptly notes, "Your SRS is a living document, meaning you will add new features and modifications with every iteration." In agile environments, requirements are often captured as user stories in a product backlog and are refined continuously throughout the project. However, the fundamental *spirit* of the SRS—the need for clear, documented, and agreed-upon requirements—persists. The form and update frequency may change, but the necessity of having a "sole source of truth" [6] for requirements becomes even more critical when changes are frequent and development is incremental.

B. **Overview of Requirements Elicitation Techniques**

Requirements elicitation is the systematic process of gathering, discovering, understanding, reviewing, and articulating the needs and constraints of stakeholders for a software system.[7] It is a critical early phase in requirements engineering, as the quality and completeness of elicited requirements directly impact all subsequent development activities. A variety of techniques are employed to effectively draw out these needs, which are often initially unclear, unstated, or conflicting. Common methods include interviews, surveys and questionnaires, document analysis, prototyping, brainstorming sessions, storyboarding, ethnography (observation), and the development of use cases.[19]

Sources [19], and [19] provide comprehensive descriptions of these techniques:

- **Interviews:** Involve direct, structured or semi-structured discussions with individual stakeholders or groups to gather detailed information, clarify ambiguities, and understand perspectives.
- **Surveys/Questionnaires:** Useful for collecting information from a large or geographically dispersed audience, often employed to gather quantitative data or broad opinions.
- **Prototyping:** Involves creating preliminary, often simplified, versions or mock-ups of the system (ranging from low-fidelity sketches to interactive high-fidelity prototypes) to allow stakeholders to visualize and interact with proposed functionalities, thereby eliciting feedback and refining requirements.
- **Document Analysis:** Consists of systematically reviewing existing documentation relevant to the project, such as business process descriptions, existing system manuals, reports, or competitor analyses, to extract pertinent requirements and understand the current context.
- **Brainstorming:** A group creativity technique designed to generate a large number of ideas or requirements in a free-flowing, non-judgmental environment.
- **Use Cases:** Focus on defining the interactions between a user (or "actor") and the system to achieve specific goals, effectively capturing functional requirements from the user's perspective through narrative scenarios.
- **Observation (Ethnography):** Involves observing users as they perform their tasks in their natural work environment to gain insights into their actual processes, pain points, and unmet needs that they might not explicitly articulate.
- **Storyboarding:** A visual technique that uses a sequence of images or panels to depict a user's interaction with the system over time, helping to explore user experience and identify key requirements.

It is important to recognize that no single elicitation technique is universally superior or sufficient for all situations. The choice and combination of techniques should be tailored to the specific context of the project, including its size and complexity, the nature and accessibility of stakeholders, the organizational culture, and the resources available.[52] As noted in [19], "Many business analysts use a combination of these techniques, as they often complement each other." For example, insights gained from document analysis might inform the questions asked during subsequent interviews, and requirements gathered through interviews might be validated or further refined using prototyping. Relying on only one method risks overlooking crucial information, missing diverse perspectives, or failing to uncover tacit (unstated) needs. A multi-faceted approach generally yields a more comprehensive and accurate set of requirements.

The following table provides a brief overview of common elicitation techniques:

Technique	Brief Description	Best For	Potential Challenges	Key Sources
Interviews	Direct discussions with stakeholders (one-on-one or group).	In-depth understanding, complex issues, clarifying ambiguities.	Time-intensive, interviewer bias, interpreting subjective data.	[19]
Surveys	Collecting data from a broad, often geographically dispersed audience.	Quantitative data, diverse perspectives, anonymity for honest feedback.	Crafting unbiased questions, low response rates, interpreting open-ended.	[19]
Workshops	Facilitated group sessions for collaborative requirement definition.	Brainstorming, consensus building, rapid idea generation.	Group dynamics, potential for dominance by some individuals.	[19] (Brainstorming)
Prototyping	Creating preliminary versions of the system for feedback.	Visualizing requirements, early validation, exploring designs.	Managing expectations (not final product), resource cost.	[19]
Document Analysis	Reviewing existing documents (manuals, reports, system specs).	Understanding current state, leveraging existing knowledge, gap analysis.	Outdated/irrelevant docs, risk of missing innovation.	[19]
Observation	Watching users perform tasks in their natural environment	Understanding context of use, uncovering tacit needs.	Time-intensive, researcher bias, scalability.	[19] (Ethnography)

	(Ethnography).			
Use Cases	Defining actor-system interactions to achieve specific goals.	Capturing functional requirements from user perspective, clear scenarios.	Can overlook NFRs, complexity in large systems.	[19]

C. **Common Challenges in Requirements Engineering**

The process of requirements engineering, encompassing elicitation, analysis, specification, validation, and management, is fraught with potential challenges. Successfully navigating these is crucial for project success. Key challenges frequently encountered include dealing with ambiguous or incomplete requirements, managing scope creep, overcoming communication barriers among diverse stakeholders, ensuring robust traceability and effective impact analysis for changes, and implementing well-managed change control processes.[4] Additionally, insufficient stakeholder involvement, particularly early in the process, overlooking or under-specifying non-functional requirements, and maintaining poor or inadequate documentation are also significant hurdles.[23]

[4] and [4] provide a detailed breakdown of several of these core challenges:

- **Ambiguous and Incomplete Requirements:** This arises when stakeholders do not fully or clearly articulate their needs, or express them in vague terms. This can lead to misunderstandings and a final product that deviates from expectations. Mitigation involves thorough communication, detailed interviews, workshops, reviews, and using standardized templates with precise language.
- **Scope Creep:** This occurs when new features, functionalities, or changes are incrementally added to a project without proper evaluation of their impact on timelines, resources, and budget. It often leads to delays and cost overruns. Mitigation requires a robust change control process where proposed changes are carefully assessed and formally approved.
- **Communication Barriers:** Miscommunication or lack of communication between stakeholders, project teams, and developers can result in errors, misunderstandings, and significant rework. This is particularly pronounced in geographically distributed teams. Mitigation strategies include utilizing collaborative tools, regular status meetings, progress reports, and formal requirement reviews to enhance transparency.
- **Traceability and Impact Analysis:** Without the ability to trace requirements

from their origin to their implementation and testing, and to understand the relationships between them, it becomes exceedingly difficult to assess the impact of proposed changes or to ensure all requirements have been adequately addressed. Implementing a requirement traceability matrix is a key mitigation technique.

- **Poorly Managed Changes:** Changes to requirements are inevitable in most software projects. However, if not managed through a defined process, they can disrupt project flow, introduce errors, and create confusion. A well-defined change management process, including procedures for requesting, evaluating, approving, and implementing changes, is essential.

Many of these challenges are deeply rooted in human factors rather than purely technical issues. Ambiguous requirements [4] often stem from unclear human communication, unstated assumptions, or differing interpretations of language. "Communication barriers" [4] are explicitly about human interaction. Even "scope creep" [4] can be driven by evolving stakeholder desires or an initial lack of clarity in expressing needs. This implies that effective requirements engineering demands not only strong technical acumen but also well-developed soft skills, including active listening, facilitation, negotiation, and conflict resolution. Issues like "Understandability" and "Implicitness" [7], where domain specialists might not articulate obvious (to them) requirements, further highlight the cognitive and communicative aspects of the elicitation process.

VII. Conclusion

A. **Synthesizing the Importance of Each Requirement Category**

This report has elucidated the distinct yet interconnected roles of various software requirement categories. **User requirements** are paramount as they capture the foundational "why" and high-level "what" from the stakeholder's and end-user's perspective, expressed in natural language to ensure shared understanding of the system's intended services and value. **Functional requirements** build upon this by detailing the specific "what" the system must *do*—its behaviors, features, and operations, such as authentication, data parsing, reporting, and system integration. **Non-functional requirements** are equally critical, defining "how well" the system must perform these functions by specifying quality attributes like scalability, performance, portability, reliability, maintainability, and availability. Finally, **System requirements**, including their subclasses of functional system requirements, non-functional system requirements, technical specifications, and interface requirements, provide the detailed technical "how"—the blueprint that guides the

architecture, design, and implementation of the software solution. Each category addresses a different level of abstraction and serves a unique purpose in the journey from an initial idea to a fully operational software product.

B. **The Foundational Role of Requirements in Delivering Valuable Software**

The systematic definition, meticulous management, and rigorous validation of all types of software requirements are not merely preliminary administrative steps in the software development lifecycle. Instead, they constitute the bedrock upon which successful, valuable, and high-quality software is conceived, designed, and constructed.[1] A diligent, expert, and comprehensive approach to all facets of software requirements engineering—from initial elicitation through detailed specification and ongoing management—is indispensable for achieving project objectives, meeting stakeholder expectations, and ultimately delivering software that provides tangible business value and a satisfactory user experience.

The entire requirements engineering process, with its various types of requirements and associated activities, is fundamentally about managing complexity and mitigating risk. By systematically decomposing abstract needs into progressively more detailed and verifiable specifications, development teams can significantly reduce uncertainty, preempt costly rework by identifying issues and misalignments early, and substantially increase the probability of delivering a software product that truly meets its intended purpose and adheres to the required quality standards. The different types of requirements explored in this report are, in essence, essential tools and frameworks in this critical endeavor of risk and complexity management within the dynamic field of software development.

Works cited

1. Defining Software Requirements for Software Selection - ArgonDigital, accessed on May 25, 2025, https://argondigital.com/blog/product-management/defining-software-requirements/
2. Why Your Project Needs Software Requirements Specification - Deviniti, accessed on May 25, 2025, https://deviniti.com/blog/application-lifecycle-management/software-requirements-specification/
3. Functional vs Non-Functional Requirements | Requiment, accessed on May 25, 2025, https://www.requiment.com/what-are-functional-and-non-functional-requirements/
4. The Biggest Challenges of Requirements Management - Visure ..., accessed on

May 25, 2025, https://visuresolutions.com/requirements-management-traceability-guide/biggest-challenges-of-requirements-management/
5. How to Write a Software Requirements Document (SRD): Best Practice, accessed on May 25, 2025, https://document360.com/blog/software-requirements-document/
6. Write a Software Requirement Document (With Template) [2025 ..., accessed on May 25, 2025, https://asana.com/resources/software-requirement-document-template
7. CS 410/510 - Software Engineering class notes, accessed on May 25, 2025, https://cs.ccsu.edu/~stan/classes/CS410/notes16/04-Requirements.html
8. Functional Requirements in Software Development: Types and B, accessed on May 25, 2025, https://www.altexsoft.com/blog/functional-requirements/
9. Nonfunctional Requirements: Examples, Types and Approaches - AltexSoft, accessed on May 25, 2025, https://www.altexsoft.com/blog/non-functional-requirements/
10. Software Requirements Specification | Ultimate Guide - QAT Global, accessed on May 25, 2025, https://qat.com/software-requirements-specifications-101/
11. System Requirements Definition - SEBoK, accessed on May 25, 2025, https://sebokwiki.org/wiki/System_Requirements_Definition
12. Functional and Nonfunctional Requirements Specification - AltexSoft, accessed on May 25, 2025, https://www.altexsoft.com/blog/functional-and-non-functional-requirements-specification-and-types/
13. Functional Requirements vs Business Requirements Explained ..., accessed on May 25, 2025, https://www.requiment.com/functional-requirements-vs-business-requirements-explained/
14. cs.ccsu.edu, accessed on May 25, 2025, https://cs.ccsu.edu/~stan/classes/CS410/notes16/04-Requirements.html#:~:text=User%20requirements,under%20which%20it%20must%20operate.
15. Comprehensive Guide to User Requirements for Software Success, accessed on May 25, 2025, https://qat.com/guide-user-requirements/
16. Functional requirement - Wikipedia, accessed on May 25, 2025, https://en.wikipedia.org/wiki/Functional_requirement
17. www.geeksforgeeks.org, accessed on May 25, 2025, https://www.geeksforgeeks.org/software-engineering-classification-of-software-requirements/#:~:text=System%20requirements%3A%20These%20requirements%20specify,a%20basis%20for%20system%20design.
18. A Template of User Requirements Document with Sample Contents - Digital Policy Office, accessed on May 25, 2025, https://www.digitalpolicy.gov.hk/en/our_work/digital_infrastructure/methodology/system_development/doc/G60c_Best_Practices_for_Business_Analyst_Appendix_C_v1_1.pdf
19. Requirements Elicitation in Software Engineering: A Complete Guide, accessed

on May 25, 2025, https://www.testbytes.net/blog/requirements-elicitation/
20. User Requirement Specifications (User Specs, URS) | Ofni Systems, accessed on May 25, 2025, https://www.ofnisystems.com/services/validation/user-requirement-specifications/
21. What are Functional Requirements: Examples, Definition, Complete Guide, accessed on May 25, 2025, https://visuresolutions.com/blog/functional-requirements/
22. What are functional requirements? In-depth guide with examples ..., accessed on May 25, 2025, https://decode.agency/article/functional-requirements-examples/
23. Functional vs Non-Functional Requirements Explained - Mad Devs, accessed on May 25, 2025, https://maddevs.io/blog/functional-vs-non-functional-requirements/
24. maddevs.io, accessed on May 25, 2025, https://maddevs.io/blog/functional-vs-non-functional-requirements/#:~:text=Here%20are%20some%20examples%20of,using%20their%20username%20and%20password.
25. What is Data Parsing and How Can You Automate it? - Klippa, accessed on May 25, 2025, https://www.klippa.com/en/blog/information/what-is-data-parsing/
26. What is Data Parsing? - Tibco, accessed on May 25, 2025, https://www.tibco.com/glossary/what-is-data-parsing
27. patco.princeton.edu, accessed on May 25, 2025, https://patco.princeton.edu/document/576
28. Example of Functional Requirements Document, accessed on May 25, 2025, https://www.businessanalyststoolkit.com/example-of-functional-requirements-document/
29. Non-Functional Requirements Capture - Engineering Fundamentals ..., accessed on May 25, 2025, https://microsoft.github.io/code-with-engineering-playbook/design/design-patterns/non-functional-requirements-capture-guide/
30. Nonfunctional Requirements Explained: Examples, Types, Tools, accessed on May 25, 2025, https://www.modernrequirements.com/blogs/what-are-non-functional-requirements/
31. NFRs: What is Non Functional Requirements (Example & Types ..., accessed on May 25, 2025, https://www.browserstack.com/guide/non-functional-requirements-examples
32. Non Functional Requirements | Quick Guide For The Business Analyst In 2025, accessed on May 25, 2025, https://businessanalystmentor.com/non-functional-requirements/
33. What are some examples of non-functional requirements? - Design Gurus, accessed on May 25, 2025, https://www.designgurus.io/answers/detail/what-are-some-examples-of-non-functional-requirements
34. Is performance a non-functional requirement? - Design Gurus, accessed on May

25, 2025, https://www.designgurus.io/answers/detail/is-performance-a-non-functional-requirement
35. Non-Functional Requirements Examples: a Full Guide - Testomat, accessed on May 25, 2025, https://testomat.io/blog/non-functional-requirements-examples-definition-complete-guide/
36. What are non-functional requirements? Detailed guide + examples ..., accessed on May 25, 2025, https://decode.agency/article/non-functional-requirements-examples/
37. Non-Functional Requirements: Definition and Examples | Glossary, accessed on May 25, 2025, https://chisellabs.com/glossary/what-is-non-functional-requirements/
38. Non Functional Requirements: Maintainability - ArgonDigital ..., accessed on May 25, 2025, https://argondigital.com/blog/product-management/non-functional-requirements-maintainability/
39. High Availability | Non-functional Requirement in System Design, accessed on May 25, 2025, https://systemdesignschool.io/fundamentals/availability
40. What are Availability Requirements?, accessed on May 25, 2025, https://requirements.com/Content/What-is/what-are-availability-requirements
41. Classification of Software Requirements – Software Engineering ..., accessed on May 25, 2025, https://www.geeksforgeeks.org/software-engineering-classification-of-software-requirements/
42. Non-Functional vs. Functional Requirements: When to Use Each Type - SPEC Innovations, accessed on May 25, 2025, https://specinnovations.com/blog/non-functional-vs.-functional-requirements-when-to-use-each-type
43. Functional vs. Non-Functional Requirements - QRA, accessed on May 25, 2025, https://qracorp.com/guides_checklists/functional-vs-non-functional-requirements/
44. How to Write a Technical Specification [With Examples] - Monday.com, accessed on May 25, 2025, https://monday.com/blog/rnd/technical-specification/
45. 10 Reasons to Write Technical Specification - Visartech Blog, accessed on May 25, 2025, https://www.visartech.com/blog/10-reasons-why-you-should-write-technical-specification/
46. Interface requirements: Overview, definition, and example - Cobrief, accessed on May 25, 2025, https://www.cobrief.app/resources/legal-glossary/interface-requirements-overview-definition-and-example/
47. External Interfaces in Software Requirements Explained - QAT Global, accessed on May 25, 2025, https://qat.com/external-interfaces-software-requirements/
48. Software Requirements Specification (SRS), accessed on May 25, 2025,

https://wildart.github.io/MISG5020/SRS.html
49. Functional vs non-functional requirements: Differences & examples - Freshcode, accessed on May 25, 2025, https://www.freshcodeit.com/blog/functional-vs-non-functional-requirements
50. Functional vs. Non-Functional Requirements - Jama Software, accessed on May 25, 2025, https://www.jamasoftware.com/requirements-management-guide/writing-requirements/functional-vs-non-functional-requirements/
51. IEEE 830-1993 - Accuris Standards Store, accessed on May 25, 2025, https://store.accuristech.com/ieee/products/vendor_id/1221
52. www.testbytes.net, accessed on May 25, 2025, https://www.testbytes.net/blog/requirements-elicitation/#:~:text=Overview%20of%20the%20Requirements%20Elicitation%20Process&text=Common%20methods%20include%20interviews%2C%20focus,and%20requirements%20of%20the%20project.
53. Understanding the Differences Between Functional and Non-functional Requirements, accessed on May 25, 2025, https://reqi.io/articles/functional-and-non-functional-requirements
54. www.perforce.com, accessed on May 25, 2025, https://www.perforce.com/blog/alm/what-are-non-functional-requirements-examples
55. Functional vs Non-Functional Requirements: Understanding the Core Differences and Examples - Ironhack, accessed on May 25, 2025, https://www.ironhack.com/us/blog/functional-vs-non-functional-requirements-understanding-the-core-differences-and
56. What Are Functional Requirements? Types and Examples - WINaTALENT Blog, accessed on May 25, 2025, https://winatalent.com/blog/what-are-functional-requirements-types-and-examples/
57. What are functional and non-functional requirements in UX design?, accessed on May 25, 2025, https://www.designgurus.io/answers/detail/what-are-functional-and-non-functional-requirements-in-ux-design