Enhancing IoT Requirements Through Layered Contextual Information

Lasse Harjumaa
Kokkola University Consortium Chydenius
University of Jyväskylä
Kokkola, Finland
e-mail: lasse.m.harjumaa@jyu.fi

Abstract— **Internet of Things (IoT) systems operate in dynamic and heterogeneous environments where behavior depends on contextual assumptions such as connectivity, data quality, and operational intent. These assumptions often remain implicit, complicating validation and traceability in requirements engineering. This paper introduces Con², a lightweight context-aware approach that integrates intent-driven context modeling with executable behavioral specifications. Con² formalizes contextual assumptions as reusable context contracts linked to Gherkin scenarios. This separation of context and behavior improves explicitness, reduces duplication, and enables automated testing and runtime monitoring. A smart building lighting case study illustrates the approach, and an analytical comparison with a conventional use case highlights improvements in clarity and verifiability while maintaining compatibility with Agile practices.**

Keywords-Internet of Things; Requirements engineering; IoT Requirements; IoT development.

I. INTRODUCTION

IoT systems differ from conventional software systems in that they operate across heterogeneous infrastructures that connect cloud services with distributed networks of sensors and actuators. By integrating hardware and software across architectural layers, they create tight coupling between physical processes and digital logic. Requirements in IoT projects rarely remain stable; many emerge only during deployment and operational use. Development must therefore accommodate technological diversity while maintaining scalability and security, which calls for methods that capture requirements in a way that reflects the dynamic and context-dependent nature of IoT systems.

Contextual information forms the foundation of meaningful system behavior. Sensor measurements cannot be interpreted reliably without considering environmental conditions and user intent, and identical inputs may require different responses depending on operational circumstances. Context is thus essential rather than supplementary. Yet contextual assumptions often remain implicit and fragmented across architectural layers, with business meaning expressed at higher levels and sensing details confined to devices. This separation makes it difficult to relate functional requirements to the conditions under which they hold. To address this gap, we propose a context-aware requirements engineering framework that embeds contextual assumptions directly into executable behavioral specifications.

IoT systems operate in changing physical environments and evolving usage scenarios, which complicates the definition of behavior in advance. Requirements are refined iteratively as real-world feedback reveals new constraints and expectations. Such conditions favor lightweight, executable specifications that evolve with the system and can be validated continuously. Behavior-Driven Development (BDD) supports this need by expressing requirements as testable scenarios that function both as documentation and as automated verification artifacts [14].

The rest of this paper is organized as follows. Section II describes related work. Section III describes the importance of context in IoT. Section IV describes our approach for including context information in requirements. In Section V, constructs and usage of our solution are described. Section VI represents case study. Section VII provides evaluation of the approach, and Section VIII discusses the findings. Section IX concludes the paper.

II. RELATED WORK

Research on context-aware computing provides foundational definitions that explain why context is essential for producing relevant system behavior. Dey's work [6] characterizes context-aware systems as those that use contextual information to deliver services aligned with a user's task. This notion of task-dependent relevance closely matches IoT scenarios, where identical measurements may lead to different interpretations or actions depending on operational purpose, such as monitoring or control. It also broadens the concept of context beyond sensor metadata by introducing an intent-driven perspective.

Within IoT, Perera et al. [11] survey context-aware computing for IoT and consolidate existing techniques and middleware support for managing context. They highlight the diversity of IoT environments and the complexity of context handling, but it focuses primarily on processing mechanisms rather than on how context should be elicited and structured as a requirement engineering artifact throughout the lifecycle.

Requirements engineering (RE) for IoT has been examined in several reviews that reveal both active research and fragmentation. A systematic mapping study by Aguilar-Calderón et al. [1] shows that most proposals address isolated aspects of RE and reflect the technical complexity of IoT systems. Similarly, da Silva Souza [5] identifies a range of RE processes and validation practices, suggesting a diverse landscape rather than a unified, end-to-end methodology.

Some efforts move toward more practical guidance. For example, RETIoT introduces structured scenarios and templates to support requirements work and reports encouraging results from project use [4]. Siakas et al. [12]

propose REFIoT, a framework that organizes IoT challenges across stakeholder and environmental dimensions to support systematic elicitation. Likewise, Boutot et al. [2] present UCM4IoT, a domain-specific modeling language that extends use cases to represent adaptive behavior across interconnected IoT components.

There are also examples of efforts to utilize Gherkin [3] language to systemize certain aspects of IoT development. Wang et al. [15] have proposed a lightweight enhancement for modeling concurrent and sequential behavior. Kannengiesser et al. [8] present an approach to address the challenges of testing Cyber-Physical Systems (CPS) by making abstractions of Gherkin test scenarios.

Despite this body of work, interest in IoT-specific RE has not significantly increased in recent years, and practice still relies largely on conventional techniques such as use cases or interviews [1]. Existing approaches remain fragmented and domain-bound, often targeting only selected phases of RE or specific applications. Given the inherently dynamic and context-dependent nature of IoT environments, more systematic methods are needed to capture contextual information and integrate it consistently into requirements across heterogeneous configurations.

In practice, context frequently remains implicit across architectural layers, which makes it difficult to connect high-level intent with low-level sensing and actuation behavior.

III. The Importance of Context in IoT

Perera et al. [11] define context as any information that characterizes an entity's situation. In software engineering, requirements are typically classified as functional or non-functional, with non-functional requirements shaping architectural and quality decisions throughout the lifecycle. In IoT systems, however, such qualities are rarely fixed. Properties like latency, reliability, energy use, or data freshness depend on operational conditions such as connectivity or device state, which makes many requirements inherently conditional rather than universally valid.

IoT applications involve diverse stakeholders who interact with different parts of a shared infrastructure. Because devices and services are interconnected across architectural layers, requirements may be interpreted differently depending on perspective. This variability increases the difficulty of specification and validation, particularly when contextual assumptions remain implicit.

When contextual knowledge is incomplete, data may become ambiguous at the application level. Although semantic metadata clarifies structure and relationships, contextual information is often only partially structured, limiting automated interpretation [13]. Misalignment between data producers and consumers can further distort meaning. In smart farming, for example, analytics-based recommendations may be disregarded when they conflict with established practices or experiential knowledge [7]. Resource-constrained devices add another layer of uncertainty by relying on external context sources that may be inconsistent or outdated [16]. Mouhim and Lachhab [10] survey contextual modeling approaches and highlights trade-offs between simplicity and expressiveness in different representation techniques.

More fundamentally, IoT requirements depend not only on inputs but also on situational context. In traditional software, identical inputs usually lead to identical outcomes. In IoT systems, the same input may produce different behavior when surrounding conditions change. Requirements therefore become adaptive and situational rather than stable and environment independent.

The layered architecture of IoT systems reinforces this dynamic. Contextual meaning emerges gradually as data moves upward through the system. A raw accelerometer reading at the device layer represents physical acceleration only; subsequent layers interpret and enrich this signal until it informs business-level decisions such as maintenance actions. Context thus accumulates across layers rather than existing as a single, fixed description. This accumulation is illustrated in Table I with an example of data from vibration sensor, which evolves from a plain acceleration value at the device level to a required action at the intent level.

TABLE I. Accumulating Context

Layer	Data	Context Type	Meaning
Device	ax,ay, az	Physical	Acceleration values
Edge	RMS, FFT	Operational	Vibration level
Cloud	Anomaly score	Analytical	Equipment health
Application	Maintenance action	Business / intent	Recommended action

IV. CON^2 Approach

We adopt Gherkin and Behavior-Driven Development (BDD) as the foundation because they align with contemporary IoT practices and reflect the event-driven nature of such systems. Gherkin provides a lightweight, executable specification language in which requirements are expressed as Given–When–Then scenarios that function as both documentation and automated tests [3]. Unlike heavier modeling approaches that remain detached from implementation, Gherkin integrates naturally with Agile and DevOps workflows and supports continuous validation. Its executable scenarios allow requirements to evolve with the system and to be verified through testing or runtime monitoring, making it a practical basis for integrating explicit context and contract semantics.

However, while Gherkin captures discrete behaviors effectively, it does not explicitly represent the continuous and distributed contextual conditions typical of IoT systems. Our extension preserves standard syntax while introducing structured mechanisms to define and reuse contextual assumptions across scenarios. By separating context from behavior, the approach reduces duplication and enables conditional requirements to be specified under clearly defined operating conditions.

A. Intent-Driven Context Modeling

The proposed context-aware requirements approach supports IoT development by making contextual knowledge explicit and structured. It begins with the systematic elicitation of operational intent and underlying assumptions so that critical dependencies do not remain implicit or fragmented across stakeholders. These are formalized as context contracts that describe the information and constraints associated with each architectural layer, reducing ambiguity and improving shared understanding. By linking contextual definitions to executable behavioral scenarios and runtime metrics, the approach establishes traceability from assumptions to observable system behavior and enables continuous validation as operating conditions evolve.

An Intent Context represents the operational purpose of the system from a stakeholder perspective. Rather than describing environmental conditions alone, it clarifies why the system operates and under which assumptions that purpose can be achieved. Intent defines which variables are relevant and what levels of performance or constraint are acceptable, thereby shaping how requirements are interpreted and how system behavior is evaluated.

Formally, we define an intent context C as the tuple

$$C = (I, S_r, S_e, Q, O, P)$$

where each element captures a distinct aspect of operational purpose and constraints. I denotes the intent statement, S_r denotes the set of required internal signals, S_e denotes the set of external inputs, Q denotes data quality constraints, O denotes operational or safety constraints, and P denotes optimization objectives or policies.

B. Contracts

Our approach draws inspiration from Design-by-Contract [9] to make assumptions and guarantees explicit and verifiable in IoT requirements. Specifications are structured around preconditions, postconditions, and invariants, which define the contextual assumptions under which behavior is valid, the outcomes the system must guarantee, and the properties that must always hold to ensure safety and correctness. In this framework, these elements are elevated from method-level assertions to system-level context contracts that define responsibilities across architectural layers. This contract-based view clarifies obligations across device, edge, and cloud components while enabling automated testing and runtime monitoring. By combining Design-by-Contract principles with executable behavioral scenarios, requirements become enforceable throughout development and operation rather than remaining purely descriptive.

Table II maps classical Design-by-Contract concepts to their counterparts in Con² and explains their role in IoT requirements engineering. Traditional elements are reinterpreted as explicit context assumptions, behavioral guarantees, and safety or quality constraints that govern when behavior is valid and what outcomes must be achieved. These constructs are organized into layered context contracts spanning architectural components to clarify responsibilities. Runtime assertion checking corresponds to continuous testing and monitoring, while contract refinement is reflected in context specialization for different operational intents.

TABLE II. CONTRACT CONCEPTS

Design-by-Contract concept	Traditional meaning	Corresponding element in Con²	Role in IoT requirements
Pre-conditions	Conditions that must hold before execution	Context assumptions / Preconditions in context contracts	Define when behavior or decisions are valid (e.g., data freshness, connectivity, calibration)
Post-conditions	Guarantees after execution	Then-clauses / Contract guarantees	Specify measurable outcomes and acceptance criteria
Invariants	Properties that always hold	Safety & quality constraints	Ensure continuous safety, reliability, and correctness
Contract	Formal agreement between caller and callee	Layered context contract (device/edge/ cloud)	Clarifies responsibilities between architectural layers
Assertion checking	Runtime verification of conditions	Tests & monitoring metrics	Continuous validation and runtime adaptation
Contract refine-ment	Stronger guarantees in subtypes	Context specialization per intent/layer	Different constraints under monitoring vs control vs optimization

A Context Contract defines the data and quality conditions associated with a given context by specifying the structure, semantics, and validity of the information on which a requirement depends. By making explicit which data must be available and trustworthy, it clarifies the assumptions underlying system behavior and addresses a common IoT issue in which data availability or quality constraints remain implicit.

A Contextual Scenario is a conventional Given–When–Then specification annotated with one or more intent contexts. The scenario inherits the context's assumptions and constraints, reducing repetition and strengthening traceability. Behavioral expectations are therefore interpreted only under explicitly defined operating conditions, which improves modularity and allows the same behavior to be evaluated consistently across alternative intents.

V. CONSTRUCTS AND USAGE

A. Language Extensions

Con² extends Gherkin with lightweight constructs that define reusable contexts and associate them with behavioral scenarios. Rather than altering the semantics of existing Feature or Scenario elements, the extension introduces additional structures that make contextual assumptions first-class and reusable. These separate the definition of operational intent and environmental constraints from behavioral descriptions, allowing requirements to be interpreted only under explicitly stated conditions.

The extension therefore augments, rather than replaces, conventional Given–When–Then scenarios. Context is defined once as structured, verifiable artifacts and then

referenced by scenarios that depend on it. This design reduces duplication, improves traceability between assumptions and behavior, and enables automated validation of both contextual conditions and functional outcomes. Together, the new constructs allow Gherkin specifications to capture not only what the system should do, but also under which contextual circumstances the behavior is valid.

A Context block declares reusable contextual assumptions and constraints that must hold for related scenarios to be valid. It encapsulates operational intent together with the required signals and quality conditions. The Intent field and contract-oriented sections - Preconditions, Invariants, Postconditions, and ExternalInputs - define contextual assumptions and guarantees outside of standard Given–When–Then scenarios.

The @context annotation associates a standard Gherkin Scenario with one or more previously defined contexts. The scenario inherits all assumptions and constraints from the referenced context. The following example illustrates the usage of these constructs.

```
Context: PredictiveMaintenance
  Intent: Early degradation detection with minimal downtime

Preconditions:
- vibration_stream freshness <= 30s
- rpm_available OR rpm_quality >= 0.9
- calibration_status == valid
- sampling_rate_hz >= 5000

Invariants:
- alarm if rms_vibration_g >= 0.80 for >= 10s
- data_completeness >= 98% /day/asset
- units: acceleration=g, frequency=Hz
- anomaly_score uses approved_model_version

Postconditions:
- health_state every <= 5 min
- anomaly_score <= 10s per window
- maintenance_recommendation includes evidence

ExternalInputs:
- asset_registry
- maintenance_history
- operating_conditions
- spare_parts_status (optional)

@context(Monitoring)
Scenario: Excessive vibration alert
  Given vibration_rms > 12 mm/s
  When sustained for 5s
  Then maintenance alert generated
```

B. Process Integration

Con^2 integrates naturally into Agile/DevOps workflows without introducing heavy upfront modeling.

Step 1. Context elicitation. Stakeholders identify operational intents and derive corresponding contexts. For each intent, required signals, external inputs, and quality constraints are documented. This step makes implicit assumptions explicit and surfaces missing information sources.

Step 2. Scenario specification. Behavioral requirements are written as contextual scenarios. Scenarios focus on decision logic and observable outcomes rather than repeating environmental details.

Step 3. Validation and testing. Contexts are automatically checked for completeness, while scenarios are executed using BDD frameworks. Context contracts guide the creation of simulation data and monitoring rules.

Step 4. Runtime monitoring. Quality constraints defined in contexts (e.g., freshness or completeness) are monitored in production, enabling continuous validation of requirements satisfaction.

C. Diverse stakeholder concerns

By aligning context contracts with both architectural layers and stakeholder perspectives, the approach provides a structured way to elicit and document requirements from multiple viewpoints without losing coherence across the system. Each layer captures the goals that are meaningful to its respective stakeholders, allowing device-level concerns such as sensing accuracy or connectivity to be specified independently from operational processing constraints or business objectives. This separation enables stakeholders to articulate requirements using familiar terminology, without needing detailed knowledge of the entire technical stack. At the same time, explicit contracts connect these viewpoints through clearly defined dependencies, ensuring that higher-level expectations remain grounded in the conditions provided by lower layers. We believe that this reduces ambiguity and improves traceability of requirements. Furthermore, it systematically integrates diverse stakeholder concerns while maintaining consistency across the IoT architecture.

VI. Case Study: Smart Building Lighting Control

To demonstrate the applicability of Con^2 in a realistic setting, we apply it to a smart building lighting control scenario derived from our ongoing project, where IoT technology is used during construction and later to monitor living conditions. The case illustrates how contextual assumptions can be elicited, formalized as contracts, and linked to executable behavioral specifications. Although compact, it reflects key IoT characteristics such as heterogeneous devices, distributed control, multiple stakeholders, and context-dependent behavior.

The system provides voice-controlled indoor lighting. Users can switch individual lights or groups through spoken commands. The solution integrates voice assistants, home automation middleware, and KNX-based building automation, spanning device, edge, and application layers.

Three stakeholder roles interact with the system: residents, presenters, and maintainers. While sharing the same infrastructure, they differ in expectations regarding responsiveness and reliability, which highlights the need for explicit contextual assumptions.

Although switching lights appears straightforward, correct operation depends on contextual conditions such as user presence, connectivity, device availability, command resolution, and timely response. In the original project documentation, these aspects were captured in a traditional use case where assumptions remained partly implicit and scattered, making validation difficult.

We therefore elicit the operational purpose as an intent context, defined here as providing reliable, low-latency voice-based lighting control inside the building. The resulting

LightingControl context is specified using Con² contract elements to make these assumptions explicit..

```
Context: LightingControl
  Intent: Provide reliable and low-latency voice-
based lighting control inside

Preconditions:
  - user_location == "inside"
  - network.connected == true
  - voice_service.available == true
  - each_light.has_unique_identifier == true

Invariants:
  - response_time <= 1s
  - device_availability >= 99%
  - authorization_valid == true

Postconditions:
  - selected_lights.state == requested_state

ExternalInputs:
  - voice_command
  - device_registry
  - KNX network
```

These elements formalize requirements that would otherwise remain implicit. For example, response time is defined as a measurable invariant, while connectivity and authorization become explicit, verifiable assumptions. Structuring requirements in this way clarifies responsibilities across architectural layers, assigning device state, command routing, and authorization control to their respective components.

Behavior is then specified using Gherkin scenarios annotated with the defined context, keeping behavioral logic concise while automatically inheriting contextual assumptions. Additional scenarios describe alternative behaviors and error conditions.

```
@context(LightingControl)
Scenario: Turn on living room ceiling light
  Given the user says "Turn on the living room
ceiling light"
  When the command is recognized
  Then the living room ceiling light shall be ON

@context(LightingControl)
Scenario: Unknown light name
  Given the user says "Turn on the red hallway lamp"
  When the device cannot be resolved
  Then an error message shall be provided

@context(LightingControl)
Scenario: Device unavailable
  Given the requested light is unreachable
  When a control command is issued
  Then the system shall report a failure
```

These scenarios focus on observable behavior, while contextual assumptions such as connectivity and latency are inherited from the context contract. This separation reduces duplication and improves maintainability, since shared assumptions are defined once. Because the scenarios follow the BDD style, they can be executed as automated tests. During development, simulated device states and voice commands verify postconditions, and in operation the same contract elements are monitored at runtime. Response time, device availability, connectivity, and command success rates are continuously tracked, allowing invariant violations to trigger alerts or fallback behavior. Requirements thus evolve from static documentation into enforceable runtime checks, supporting continuous validation within DevOps workflows.

Applying Con² to the lighting scenario demonstrates practical benefits. Contextual assumptions become explicit and testable, separation of context and behavior reduces repetition, layered contracts clarify responsibilities, and executable specifications enable automatic validation of both functional and quality requirements. Even in this compact example, the dependency of behavior on contextual factors becomes evident, reinforcing the value of making context explicit to improve clarity and maintainability.

VII. EVALUATION

We evaluate the proposed Con² approach using criteria relevant to IoT requirements engineering: explicitness of assumptions, traceability between context and behavior, and executability. Rather than conducting a large-scale empirical study, the goal is to analytically assess whether the approach improves clarity, structure, and verifiability compared with the conventional format used for the smart building use case.

The comparison contrasts two representations of the same functionality, Lighting-1: the original textual use case and the Con²-based specification built from intent contexts, contracts, and contextual scenarios. Although both describe voice-controlled lighting, they differ in how contextual knowledge is represented and validated. The analysis examines which assumptions are implicit or explicit, and how clearly responsibilities are assigned across architectural layers.

In the traditional use case, preconditions, postconditions, and interaction steps are presented narratively, leaving many contextual aspects informal. Network availability, device readiness, response time, and authorization appear as textual assumptions without structured support for validation or reuse, which increases the risk of being overlooked. In contrast, the Con² specification separates context from behavior by formalizing assumptions and quality constraints as contracts while scenarios describe observable outcomes. This structure enables reuse across scenarios and supports automated testing and runtime monitoring, turning contextual statements into verifiable system properties.

Table III summarizes the differences using concrete elements drawn directly from the original use case. The comparison suggests three main benefits. First, Con² improves explicitness by converting informal assumptions into structured, contract-based elements. Second, it enhances traceability by linking contextual conditions directly to executable scenarios. Third, it increases verifiability by enabling automated testing and runtime monitoring of both functional and non-functional requirements.

VIII. DISCUSSION

Con² augments behavioral specification with contextual semantics. With structured yet lightweight constructs for context and contracts, it enhances explicitness without imposing substantial overhead. Improved explicitness is the central benefit observed in the case study, which also improves traceability. Each contextual scenario links high-level intent to an executable behavior, making it possible to reason systematically from stakeholder goals to concrete tests.

In IoT projects, requirements frequently emerge only after deployment and operational feedback. Contexts and scenarios can evolve alongside the system, enabling gradual refinement rather than heavy upfront modeling. Traditional requirements techniques often assume stable environments and may struggle to capture the dynamic behavior of IoT systems.

TABLE III. TABLE TYPE STYLES

Aspect from the original use case	Conventional use case representation	Con² representation	Practical effect
User must be inside building	Informal precondition text	Explicit precondition in context contract	Can be validated using location sensing
Network must be operational	Mentioned as assumption	Context invariant (connectivity == true)	Enables automatic monitoring and alerts
Devices must be ready	Narrative statement	Device availability invariant	Clear responsibility for device layer
Unique light identifiers	Implicit design constraint	Required internal signal	Improves device registry traceability
Response time ≤ 1 s	Non-functional note	Measurable quality constraint	Testable in CI and runtime
Authorization required	Textual rule	Operational constraint	Enforceable and auditable
Voice command behavior	Step-by-step scenario	Gherkin contextual scenario	Executable automated test
Error handling	Listed exceptions	Separate contextual scenarios	Testable failure paths
Shared assumptions across scenarios	Repeated or implied	Defined once in context block	Reduces duplication
Responsibility across layers	Not explicit	Layered context contracts	Clear device /edge/cloud roles

Although demonstrated using a smart building scenario, the principles underlying Con² are not domain specific. Any IoT system where behavior depends on environmental or operational conditions may have advantage from explicit context modeling. However, for small or static systems, the potential modeling overhead may not be justified. The benefits become more pronounced as the system complexity, heterogeneity, or stakeholder diversity increases.

The current evaluation remains qualitative and limited to a single illustrative real-world case, which limits the ability to generalize the findings across diverse domains and project scales. Furthermore, the assessment of clarity and traceability relies on analytical comparison rather than strictly controlled empirical measurements, which may introduce subjective interpretation. Broader empirical studies are required to assess usability of the approach, and measurable impacts on clarity, traceability, and validation of requirements in IoT.

IX. CONCLUSION AND FUTURE WORK

IoT systems operate in dynamic and context-dependent environments where behavior depends not only on functional logic but also on implicit environmental and operational assumptions. This paper introduced Con², a lightweight context-aware requirements engineering approach that makes such assumptions explicit through intent-driven context contracts and executable behavioral scenarios. By integrating contextual modeling with Gherkin-based specifications, Con² improves explicitness while remaining compatible with Agile and DevOps practices. The exemplar case study indicates that even simple IoT application scenarios benefit from separation of concerns, supporting the argument that context should be treated as a first-class element in IoT requirements engineering. Future work will focus on empirical evaluation and tool support in larger industrial deployments.

REFERENCES

[1] J.-A. Aguilar-Calderón, C. Tripp-Barba, A. Zaldívar-Colado, and P.-A. Aguilar-Calderón, "Requirements engineering for Internet of Things (IoT) software systems development: A systematic mapping study," IEEE Access, vol. 12, no. 15, p. 7582, 2022.

[2] P. Boutot, M. R. Tabassum, A. Abedin, and S. Mustafiz, "Requirements development for IoT systems with UCM4IoT," Journal of Computer Languages, vol. 78, p. 101251, 2024.

[3] Cucumber Ltd., "Gherkin Reference Documentation," Accessed: Feb. 14, 2026. [Online]. Available: https://cucumber.io/docs/gherkin/.

[4] D. V. da Silva, B. P. de Souza, T. G. Gonçalves, and G. H. Travassos, "A requirements engineering technology for IoT software systems," arXiv, 2021.

[5] L. da Silva Souza, F. B. Ayres, P. H. T. Costa, and E. D. Canedo, "Requirements engineering processes in the context of IoT and requirements validation techniques," in Proc. WER, 2022.

[6] A. K. Dey, "Understanding and using context," Personal and Ubiquitous Computing, vol. 5, no. 1, pp. 4–7, 2001.

[7] A. Kamilaris, A. Kartakoullis, and F. X. Prenafeta-Boldú, "A review on the practice of big data analysis in agriculture," Computers and Electronics in Agriculture, vol. 143, pp. 23–37, 2017.

[8] U. Kannengiesser, F. Krenn, and C. Stary, "A behaviour-driven development approach for cyber-physical production systems," in Proc. 2020 IEEE Conf. on Industrial Cyberphysical Systems (ICPS), 2020, pp. 179–184.

[9] B. Meyer, "Design by Contract," in Advances in Object-Oriented Software Engineering, Prentice Hall, 1991, pp. 1–50.

[10] S. Mouhim and F. Lachhab, "Towards a context awareness system using IoT, AI, and big data technologies," IEEE Access, vol. 13, pp. 40302–40315, 2025.

[11] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the Internet of Things: A survey," IEEE Communications Surveys & Tutorials, vol. 16, no. 1, pp. 414–454, 2014.

[12] E. Siakas et al., "REFIoT: A framework to combat requirements engineering in IoT applications and systems," in Systems, Software and Services Process Improvement, vol. 2179, Springer, 2024, pp. 80–96.

[13] M. Simpson et al. "A platform for the analysis of qualitative and quantitative data about the built environment and its users," in Proc. 2017 IEEE 13th Int. Conf. on e-Science, 2017, pp. 228–237.

[14] C. Solis and X. Wang, "A study of the characteristics of behaviour driven development," in Proc. 37th EUROMICRO Conf. on Software Engineering and Advanced Applications (SEAA), Aug. 2011, pp. 383–387.

[15] B.-Y. Wang, Y.-C. Yen, and Y.-C. Cheng, "Specifying Internet of Things behaviors in behavior-driven development: Concurrency enhancement and tool support," Applied Sciences, vol. 13, no. 2, p. 787, 2023.

[16] N. Zubair, N. A., K. Hebbar, and Y. Simmhan, "Characterizing IoT data and its quality for use," arXiv, 2019.