Journal of Universal Computer Science, vol. 32, no. 8 (2026), 1103-1132
submitted: 20/6/2025, accepted: 12/1/2026, appeared: 28/8/2026 CC BY 4.0

MicroCEP: Complex Event Processing on the Edge

Guadalupe Ortiz
(University of Cadiz, Puerto Real, 11519, Spain
https://orcid.org/0000-0002-5121-6341, guadalupe.ortiz@uca.es)

Javier Tena
(University of Cadiz, Puerto Real, 11519, Spain
https://orcid.org/0009-0008-3796-7916, javier.tenawilson@alum.uca.es)

Adrian Garcia-Bermejo
(University of Cadiz, Puerto Real, 11519, Spain
a.garber@alum.uca.es)

Alfonso Garcia-de-Prado
(University of Cadiz, Puerto Real, 11519, Spain
https://orcid.org/0000-0002-6523-9974, alfonso.garciadeprado@uca.es)

Stephan Reiff-Marganiec
(University of Derby, Derby, DE22 1GB, United Kingdom
https://orcid.org/0000-0001-6456-7063, S.Reiff-Marganiec@derby.ac.uk)

Abstract: The Internet of Things (IoT) has experienced rapid growth, taking advantage of advanced data processing. While complex event processing (CEP) engines have positioned themselves as a good solution for cloud and fog data processing, there are no CEP solutions designed to run on microcontrollers (MCUs) at the edge. This paper presents MicroCEP, a software architecture for MCUs that enables real-time filtering and correlation of sensor data, both locally obtained and received through message queues at the edge. MicroCEP provides (i) a lightweight CEP architecture for MCUs, (ii) a pattern language to specify complex event patterns, (iii) a REST API for dynamic threshold updates, and (iv) a methodology for designing and deploying MicroCEP-based systems. The solution facilitates the detection of situations of interest and the triggering of actions directly from the MCUs, enabling autonomous decision-making close to the sensors. A proof of concept implementation using RISC-V MCUs provides evidence that MicroCEP can deliver CEP capabilities at the edge on resource.constrained devices.

Keywords: Complex Event Processing, Internet of Things, Edge Computing, Microcontrollers
Categories: C.3, H.3.4, K.6.3
DOI: 10.3897/jucs.162705

1 Introduction

The Internet of Things (IoT) has experienced exponential growth in recent years, driven by advances in technologies, communications, and streaming data processing. Applications of such data processing encompass a wide range of domains and significantly enhance decision-making in various fields, primarily due to the ability to

capture and correlate heterogeneous data from multiple information sources [Zhang and Tao, 2021] and to act rapidly on the derived insights. Most solutions in this domain primarily focus on cloud-based processing with high data throughput, or on intermediate elements in the fog and gateways, which, although not possessing the same processing resources as a cloud infrastructures, offer sufficient capacity to host powerful software capable of handling a high rate of events per second [Kimovski et al., 2021]. Many of these proposals have successfully demonstrated the capabilities of Complex Event Processing (CEP) to handle heterogeneous streaming data with high throughput and low latency [Bazan-Muñoz et al., 2024, KrishnA et al., 2024, Ortiz et al., 2022]. Indeed, CEP is a technology that facilitates the correlation of large volumes of streaming data (events), enabling the detection of predefined situations of interest through what are known as event patterns. These event patterns define the expected sequence of events —combinations and temporal relationships among events— necessary for a situation to be recognized. Consequently, the CEP engine operates by continuously verifying whether incoming events conform to the sequence specified within the pattern. Depending on the breadth of the grammar provided by the CEP language utilized, patterns of varying complexity can be defined.

On the other hand, the use of low-cost microcontrollers (MCUs) has gained significant prominence in the IoT domain, enabling a paradigm known as edge computing. It should be noted that the term edge computing is subject to diverse interpretations. While it generally refers to processing close to the IoT sensors and actuators, some authors describe it in layers closer to the fog, on devices with greater processing capacity (such as a Raspberry Pi) [Kimovski et al., 2021], while other authors consider it to be located on devices with much lower processing capacity, such as an MCU where the sensors, whose information is being processed, are located [Sudharsan et al., 2022a]. In this paper, when we mention edge computing, we adopt this latter perspective and we focus on the the processing of IoT data directly on MCUs.

Despite the increasing use of MCUs for edge computing, there is a shortage of proposals that facilitate the correlation of data originating from diverse sources in such devices. While there are proposals that process isolated data at the edge or can perform simple correlations of data sensed by multiple sensors on a single edge device, far fewer proposals allow communication between several low-resource devices at the edge and the correlation of data from several of these devices in one of them in order to make real-time decisions autonomously without relying on a supervisory node with greater processing capacity. Furthermore, to our knowledge, there are no general-purpose CEP engines that can be executed in this type of MCU. The only closely related work is µCEP, which brings a restricted CEP rule language and engine that it is not open source and, in addition, does not address how multiple MCUs could collaborate in decision-making. This gap motivates the need for an open and more expressive and systematic CEP solution explicitly designed for resource-constrained MCUs and scenarios of collaboration.

To address these challenges, this paper presents MicroCEP, with the following main contributions. Firstly, we have designed a software architecture for low cost MCUs that enables (i) filtering and correlating data acquired either directly from sensors connected to the board or received via messaging queues, (ii) processing of such data in real-time by verifying compliance with a set of established conditions, and subsequently (iii) transmitting relevant data or alerts to other MCUs or external systems. Secondly, we have specified a language suitable for deployment to MCUs

which is used for the definition of event patterns for detecting and acting on situations of interest. Thirdly, we have provided a REST API that permits runtime updates of the thresholds defined within the event patterns deployed to MCUs. Lastly, we have proposed a methodology to facilitate the use of the software architecture proposed as well as the language to define the event patterns to be evaluated at runtime in an MCU within the required scenario. Unlike prior lightweight CEP proposals such as μCEP, MicroCEP combines an explicit architecture, a more expressive pattern language, a runtime reconfiguration interface and support for collaborative MCU networks in a single, open and MCU-oriented framework. The proposed solution has been implemented in C++ and successfully tested on a set of MCUs with an open-source RISC-V architecture.

The rest of the paper is organized as follows. Section 2 presents the background on the main concepts of CEP as well as MCUs and their programming. Afterwards, related work in the domain of CEP for MCUs is analyzed in Section 3. Then, the proposed software architecture is described in Section 4. Next, Section 5 presents the language for CEP patterns and their deployment to MCUs and illustrates it with several typical examples. Afterwards, the methodology to be followed to obtain benefit from this proposal is explained through a proof-of-concept implementation in Section 6, followed by a performance evaluation in Section 7. Finally, the discussion is presented in Section 8 and Section 9 presents the conclusions and comments on future work.

2 Background

This section briefly introduces CEP and provides background knowledge on MCUs.

2.1 Complex Event Processing

CEP is a technique used in data analytics to identify meaningful patterns within real-time or near-real-time event streams. Events in this context refer to data that is detected or obtained from a specific system, such as data from a sensor, a social network, or any other data stream. They are referred to as simple events. CEP patterns, on the other hand, are a series of conditions that incoming events must meet for the system to detect a specific situation of interest within the application domain. When a pattern is met, a complex event is raised (i.e., an event that represents not an isolated value, but a detected condition).

To analyze and correlate the incoming event stream using the defined patterns, a CEP engine is used. Each CEP engine has a specific Event Processing Language (EPL) for defining the patterns. The outputs of the engine can feed back into the system, as if they were new simple events in the incoming data stream.

There is a wide variety of CEP engines available, for example, Esper CEP [EsperTech, 2025], Siddhi [WSO2, 2019], or Drools [Apache, 2025]. They are all used in various commercial and research applications with slight differences between them leading to respective advantages and disadvantages. What they all have in common is that current CEP engines require devices with considerable resources for their installation, and they all necessitate a Java runtime environment. To the best of our knowledge, there is neither a commercial nor an open-source CEP engine for MCUs available.

2.2 Microcontrollers

Nowadays, MCUs are the basis of countless devices, from household appliances to industrial systems. These small integrated circuits are capable of combining a processor, memory and various peripherals to execute specific tasks with precision and efficiency. Unlike a traditional computer, which is designed for multiple purposes, an MCU focuses on specific operations, making it ideal for automation, sensor control and real-time processing.

The popularity of MCUs is due to their low power consumption, low cost and the possibility of integrating them into almost any application without the need for large infrastructures. They are capable of operating in autonomous systems, responding quickly to environmental stimuli without relying on complex operating systems. This makes them essential tools for engineers and developers seeking to create efficient solutions. Depending on the MCU model, they can interact with sensors, displays, motors and other devices, making it possible to develop systems ranging from small control systems to more advanced projects in the industrial, medical or IoT fields.

To program these devices, C and C++ are the most commonly used languages, although others such as Python or even assembly are also supported. Arduino IDE, PlatformIO or Keil uVision are common development environments that facilitate the writing and deploying of the code onto the MCU.

A very popular device is the Espressif ESP32-C3 [Espressif, 2025], which is based on the RISC-V architecture and provides about 400KB of SRAM. RISC-V is an efficient open alternative to proprietary architectures. It offers a balance between performance, low power consumption and connectivity, making it an ideal choice for IoT and automation projects. Despite its low cost, the ESP32-C3 integrates Wi-Fi and Bluetooth Low Energy (BLE), making it easy to create smart devices without the need for additional components. In addition, its compatibility with environments and frameworks such as Arduino and ESP-IDF allows both beginners and professionals to easily take advantage of its potential.

3 Related Work

As mentioned in the introduction, numerous studies have employed CEP for the detection of events of interest within the IoT. A variety of CEP engines have been utilized for this purpose, with Esper CEP [Bazan-Muñoz et al., 2024, Ortiz et al., 2022], Siddhi [Ma et al., 2015, Rosa-Bilbao et al., 2023] and Drools [Lan et al., 2019] being the most widely adopted. Many proposals also highlight and promote mechanisms for performing CEP at the edge, such as the authors in [Lan et al., 2019], but end up doing the pattern computation on machines with significant resources; in this case, for example, on an Intel Core i5. We can also highlight [Kharboucha et al., 2018], which introduces an innovative system for early detection of hypoglycemia episodes in diabetic patients. This system is based on the application of CEP techniques, which enable real-time analysis of data collected by a glucose sensor. The system architecture comprises two main components: an ESP32 MCU and a Raspberry Pi computer. The ESP32 is only responsible for acquiring data from the glucose sensor and transmitting this data to the Raspberry Pi, where the CEP processing takes place.

Another relevant line of work is EdgeCEP [Choochotkaew et al., 2017], which proposes a fully distributed CEP framework for smart IoT edge devices collaborating

in self-organizing ad hoc networks. Its main contribution is a CEP engine capable of executing processing tasks in a decentralized and optimized manner over dynamic edge topologies, combining the advantages of traditional event-based CEP with data-stream processing models. EdgeCEP introduces a hybrid model with a pseudo-source mechanism that both broadens the range of supported processing scenarios and obviates the need for explicitly specifying event sources. In addition, it defines a new event-specification language that supports relational-style operations over events. However, although EdgeCEP is positioned at the edge, its target devices have significantly higher processing capabilities than typical MCUs, and therefore it does not directly address the resource-constrained platforms considered in this work.

Beyond these approaches, another relevant contribution is a proposal that introduces a lightweight CEP engine based on the Concurrent Reactive Objects (CRO) model [Pietrzak et al., 2012]. The core technique relies on compile-time translation of CEP queries into a CRO program for resource-constrained embedded platforms. For pattern specification, the authors adopt the CEDR language, a declarative query language for event streams in which each query captures both event types and their temporal ordering. CEDR provides operators such as *All, Any, Sequence* (ordered occurrence), and *Unless* (absence within a given time window), together with value-based filtering through *When* and event correlation through *Where.* During evaluation, a pattern is maintained in one of four states: *Ignore* (not yet activated), *Await* (partial match, waiting for more events), *Accept* (pattern successfully matched), or *Fail* (the pattern has definitively not occurred). However, the authors do not report detailed performance metrics, such as measured throughput or processing latency, which limits the quantitative assessment of the proposed engine.

The fact is that even though several research efforts have explored approaches towards implementing lightweight CEP engines on the edge, no lightweight CEP system has yet been demonstrated to run effectively on MCUs with the exception of µCEP [Ren et al., 2021]. In this context, Ren et al. specifically target industrial IoT scenarios in which CEP is executed directly on MCUs rather than on more powerful edge or cloud nodes. This approach proposes the use of simple CEP rules deployed on an Arduino board. More concretely, µCEP is implemented in C/C++ and evaluated on Arduino Nano 33 BLE boards equipped with an Arm Cortex-M4 CPU running at 64 MHz, 256 KB of RAM and 1 MB of flash memory. The semantics of this proposal is very limited, and some of their proposed rules are not part of the proposed CEP language as stated by the authors at the end of section 5.1 in [Ren et al., 2021]. The rule language is derived from ETALIS and supports runtime reconfiguration of rules; however, its expressiveness remains restricted (only some operators from ETALIS are accepted in the MCU), particularly with respect to temporal windows and more complex timing constraints. The same group of authors contributes to a broader ecosystem around this proposal, including ETALIS for stream reasoning and CEP, TinyOL for TinyML with online learning on MCUs, and µCEP as the CEP engine for MCUs. However, it deserves special mention that this proposal integrates neural networks in the MCUs through TinyML [Sudharsan et al., 2022b], which is a highly relevant contribution, although outside the scope of this paper. This integration is realized in combination with TinyOL-style online learning techniques, enabling on-device model adaptation. It is also noteworthy that in order to avoid having to program the rules by hand, the authors of the µCEP engine provide the option to specify the rules in Node-RED and automatically generate the µCEP code from the Node-RED flow

[Ren et al., 2022]. The grammar provided by Node-RED for the rules is very limited, and it cannot represent complex temporal constraints (the example given in the paper is a comparison of the type "greater than"). Consequently, only relatively simple threshold-based rules and patterns can be specified through this interface. From a performance perspective, the authors report throughput values of up to 2000 events per second, which they argue is more than sufficient, since a typical sensor rarely produces more than 200 events per second. However, performance tests have been carried out internally on the MCU, measuring only the loop execution of the pattern itself, rather than reading data from sensors or receiving data from outside external sources, which would constitute a much more realistic evaluation scenario for the tests. They also observe that the throughput decreases as the number of rules added to the engine grows, and that this degradation is not linear, because not all rules impose the same computational resources. In summary, µCEP is the only proposal for a CEP engine on MCU that we have found, but it provides a very limited language for expressing rules and does not provide a reusable software implementation for processing the rules defined with their language. To the best of our knowledge, the µCEP engine for MCUs described in [Ren et al., 2021] is not available as an open-source implementation. Nevertheless, both µCEP and MicroCEP have been implemented in C++, in contrast to other proposals.

Unlike µCEP, our proposal, MicroCEP, not only provides a language for CEP programming, but also the software and methodology to process CEP patterns in MCUs and to facilitate collaboration between MCUs; it is open source and available to the whole scientific and software development community[1]. MicroCEP allows for the definition of complex patterns as needed in many real-world applications. We also

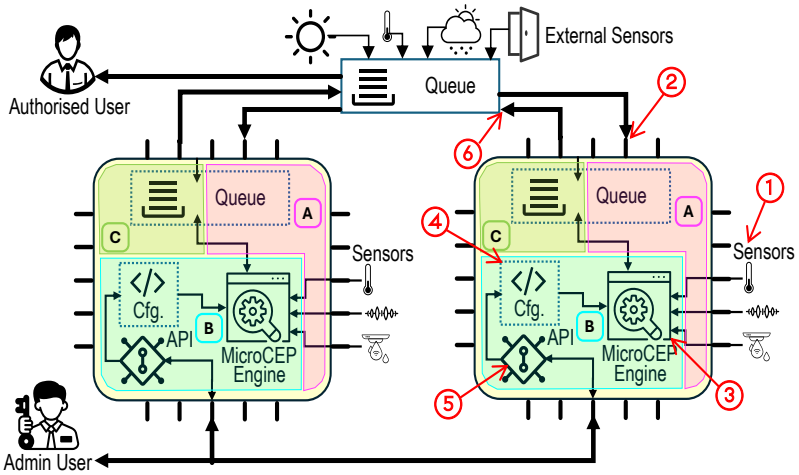

Figure 1: MicroCEP software architecture

[1] MicroCEP source code us available at https://gitlab.com/ucase/public/microcep under CC-BY licence. The case study and experimental data referred in this paper is also available at Mendeley Data: https://data.mendeley.com/datasets/r3bxphvtkk/1 (DOI: 10.17632/r3bxphvtkk.1)

provide the means to update thresholds used in the patterns at runtime, adding additional robustness at runtime.

4 MicroCEP Software Architecture

The MicroCEP software architecture is represented in Figure 1 and is composed of the following elements:

The **input module** (A) comprises the software elements responsible for managing simple events to be processed. It consists of software for handling events obtained from sensors connected directly from the MCU (see number 1 in the figure) as well as events posted from external sensors (2). The latter are received via the MQTT protocol through a Mosquito queue.

The **processing module** (B) provides the software elements that process data from the input module, detect events of interest according to patterns defined in the MCU, and raise the corresponding complex event. It is composed of the following elements:

- MicroCEP engine (3): this component processes data received from sensors and performs evaluations based on predefined rules to detect complex events in real time.
- Configuration (4): this component stores and manages system configurations based on the MCU's configuration file, including rules for event detection.
- REST API (5): this component provides an interface for updating thresholds in any condition of the pattern configuration file. The API expects to receive the required data for the updates and responds with an HTTP status code.

The **output module** (C) comprises the software elements that take care of the complex events detected by the MicroCEP engine. Such complex events may require interacting with an actuator or sending the relevant information to other MCUs or similar external systems through MQTT (6).

5 MicroCEP Pattern Language and Device Deployment

We have discussed how patterns form the key concept in CEP that leads from simple events to complex events in a timely fashion. In our framework patterns are deployed to the MCU for use in the processing module. This section introduces the overall semantic and runtime model used by MicroCEP to evaluate patterns, and then discusses the needs for a language to describe patterns and how they are deployed to the MCU.

Each pattern involves defining a series of conditions that, when met, result in a complex event derived from one or more simple or complex events. Hence, the pattern recognition language needs to allow for the generalized definition of patterns for various types of sensors or data sources and be reusable across different application domains. At runtime, MicroCEP evaluates patterns over a continuous stream of simple events produced by sensors attached to the MCU and/or received via input message queues. Each incoming measurement cycle updates an internal data map that stores the latest value observed for each sensor or data source (identified by a sensor_code), and patterns are evaluated against this data map according to a common semantic and runtime model.

Patterns are defined in a configuration file, deployed to the MCU via the built-in SPIFFS file system, and interpreted during execution. A REST API allows threshold

values in these files to be modified at runtime. From a semantic perspective, a pattern is defined as a set of condition groups. Each group contains one or more rules, and groups are combined disjunctively: a pattern is satisfied if at least one group is fulfilled. Within a group, rules are combined conjunctively: all rules in the group must evaluate to true for the group to be satisfied. When a pattern becomes satisfied, the corresponding action is triggered (e.g., sending an output message or activating an actuator), and the pattern state is reset so that a new evaluation cycle can start.

The temporal behaviour of a pattern is controlled at two levels. At the pattern level, where *mode* and *time_cycles* parameters define the global evaluation window, either in time (milliseconds) or in a discrete number of measurement cycles. Depending on the selected mode, the pattern may require all conditions to be satisfied within a fixed time window, across a given number of consecutive cycles, within a single cycle, or without any explicit temporal bound; if no group becomes satisfied before the corresponding window expires (where applicable), the pattern is reset without firing. At the rule level, each condition is associated with its own evaluation mode and timing parameters, which determine how often it is evaluated and for how long its value remains true, when applicable. Finally, the evaluation order within each group can be influenced by the check flag associated with each rule: when check is enabled and a rule evaluates to false, the remaining rules in that group are not evaluated in that cycle, which both reduces computational effort and may enforce a sequential precedence among conditions within the same group.

Before moving into defining the language and giving some more detailed examples, it will be helpful to understand the kind of scenarios that we need to capture in the CEP patterns.

Example 1: A temperature sensor shows that temperature rises above a threshold and once it exceeds that we do not wish to consider additional readings in the next few minutes (maybe because we know that a counteraction will take several minutes to take effect).

Example 2: A greenhouse measures humidity at regular intervals; the plants will take damage if the humidity is below a threshold for at least 3 cycles.

Example 3: A patient monitor records abnormal heart rate combined with a change in temperature and blood oxygen levels simultaneously.

As we can see from the examples, each pattern has one or more sensors or data readings that it talks about as well as information on timings or reading cycles. The examples are quite simple, but it is easy to imagine more complex ones as they generally put one reading in the context of others. Overall, we need ways to express simple patterns for a specific sensor and then for combining the patterns; we also need ways to express constraints that apply to an overall group of patterns.

The MicroCEP configuration file is structured into three main parts. First, the input/output configuration specifies how the MCU interacts with its environment, including the definition of data sources (sensors or input queues) and sinks (actuators or output queues), as explained in Section 5.1. Second, the pattern general configuration defines the global timing behaviour of the pattern and how often its conditions are evaluated, as described in Section 5.2. Finally, the pattern conditions section enumerates the individual rules that compose the pattern, organized into groups that determine how simple events are combined to detect complex situations, as detailed in Section 5.3.

5.1 Input/Output Configuration

At the beginning of the configuration file we need to define a set of variables that specify, among other aspects, the sensors from which data are read and/or the input queues from which events are consumed, as well as the actions to be taken when the pattern is satisfied (actuators to be triggered and/or information to be sent to an output queue). These variables include:

in_mqtt_server: IP address of the broker from which input messages are received.

in_mqtt_topic: Topic to which the client is subscribed.

in_mqtt_client_id: Unique client identifier within the input broker.

o_mqtt_server: IP address of the broker to which output messages with detected complex events are published.

o_mqtt_topic: Topic under which output messages with detected complex events are published.

o_mqtt_client_id: Unique client identifier within the output broker.

variableToSend: It determines the content that will be published in the message payload. Published messages consist of a text string and MicroCEP currently supports two alternatives: (i) including a sensor code and the value that triggered the complex event, separated by a comma (for instance, sensor_code: temp, value: 34); or (ii) including a literal string predefined by the user, such as Fire Alert. When the receiver obtains the message, it handles or stores the received text according to its own application logic.

5.2 Pattern General Configuration

The general information from the pattern determines the 'timing' of evaluations and allows quick access to an overall outcome of all rule in the pattern. A pattern primarily falls into 1 of 5 groups which capture how quickly the embedded conditions need to be satisfied. These groups are identified by a `mode` (type byte) with value 1 through to 5 followed by a number of parameters:

mode, time_cycles

The parameters capture the following, as summarized in Table 1:

`mode`: We will have five different modes of operation as explained afterwards. The following applies to all modes: (i) once a pattern is satisfied, both the pattern itself and its expressions are reset; (ii) for a pattern to be satisfied, at least one of its condition groups must be fulfilled and (iii) for a condition group to be fulfilled, all conditions within the group must be satisfied.

`time_cycles`: its value is required for mode 1 (it will then measure time), and for modes 2 and 5 (it will measure cycles). For modes 3 and 4 it is not necessary and the value will be ignored.

- In mode 1: it is used to define the temporal window (time) during which the conditions within the pattern must be fulfilled. The value of time specifies the maximum duration allowed for the set of conditions to become true. If the required conditions are not met within this defined time window, the pattern automatically resets its internal state, thus allowing a new evaluation cycle to begin. In the event that the pattern is satisfied before the configured time limit, the action associated with the pattern is executed and then the pattern is reinitialized —resetting its state and timers—to begin a new evaluation cycle.

This behavior ensures that time-bound events are detected with predictable latency and that incomplete evaluations do not persist.

- In modes 2 and 5: it provides an analogous purpose but operates within a discrete, cycle-based context (cycles). Instead of expressing the constraint in seconds or milliseconds, it defines the number of measurement cycles during which the conditions must be fulfilled. This approach is particularly suitable for systems where data acquisition is periodic. Currently the cycles are internally set to 5000ms; in the future we might include a variable with which the user can also decide the duration of the cycle.

Parameter	Description
mode	Defines the operational mode of the pattern (values 1–5).
time_cycles	Defines the duration (in milliseconds) during which all conditions must be satisfied for mode1; the number of measurement cycles available for the conditions to be satisfied in modes 2 and 5; a placeholder for modes 3 and 4.

Table 1: Parameters for pattern general configuration

The five operating modes of the pattern are explained below and summarized in Table 2:

Mode 1, known as by time, establishes a fixed time interval within which the conditions specified by the pattern must be satisfied. If the event conditions are not fulfilled within this interval, the pattern ends without being fired, and all expressions are reset. This mode is commonly used for detecting transient phenomena that must occur within a strict temporal limit, such as detecting a temperature rise above a threshold within a period of time.

Mode 2, referred to as by consecutive cycles, specifies a number of consecutive measurement cycles during which the conditions must evaluate to true for the pattern to be activated as true. This mode is used when persistence across repeated readings is required, for example, to confirm a stable anomaly that is not caused by a single outlier.

Mode 3, called single cycle, expects all conditions within the pattern to occur simultaneously within one evaluation cycle: if the pattern is not satisfied, all its expressions are reset. It is the most restrictive mode in terms of timing, as it only triggers when all relevant sensor readings align at the same moment. When the pattern specifies a number of cycles and a condition inside the pattern also specifies a number of cycles, the total number of cycles required is the product of the two.

Mode 4, or indefinite time, relaxes all temporal constraints, allowing the pattern to accumulate partial conditions over an extended or undefined duration. Once all individual conditions have been satisfied—regardless of when—the pattern is marked as true. This mode is particularly useful for detecting long-term or slowly developing events.

Mode 5, termed by cycles, operates in the same way as Mode 1; however, instead of working in milliseconds, it functions based on cycles. In other words, time windows are initiated for a specified number of cycles, and once this number of cycles is completed, all expressions are reset. This mode is useful when events must occur within a fixed number of measurement cycles rather than a specific time interval, for example, when monitoring processes synchronized with sensor readings.

Mode	Name	Required Parameters	Description
1	By Time	mode, time_cycles (time)	The pattern must be fulfilled within a defined time window. If not, the pattern resets.
2	By Concecutive Cycles	Mode, time_cycles (cycles)	The pattern must be satisfied across a defined number of consecutive measurement cycles.
3	Single Cycle	Mode	All conditions must occur simultaneously within a single evaluation cycle.
4	Indefinite Time	Mode	The pattern remains valid indefinitely until all conditions are fulfilled, regardless of time.
5	By cycles	Mode, time_cycles (cycles)	The pattern must be fulfilled within a specified number of cycles.

Table 2: Pattern operating modes

5.3 Pattern Conditions

Each pattern contains a number of rules which specify the conditions on individual events and the timings in which these need to arrive. We call a set of rules a condition block and a pattern can contain one or more condition blocks. Rules in one block are conjunctive, blocks in a pattern are disjunctive.

Each rule follows this syntax; the parameters are explained below and summarized in Table 3:

mode, time_cycles, group, check, sensor_code condition value

Please note that the parameters have to be provided in this order, separated by commas and without line breaks. As we can see, each rule in the condition block has a mode, with a time-cycle parameter as we have seen in the overall rule. A pattern may define multiple rules, which are partitioned into groups via the group identifier (rules belonging to the same group share the same group value). As previously mentioned, within a group, rules are combined conjunctively (logical AND): every rule in that group must evaluate to true for the group to be satisfied. Across different groups, the engine combines results disjunctively (logical OR): the pattern is satisfied if at least one group evaluates to true. Rules are evaluated in order of occurrence and would normally be all evaluated assuming the preceding one was true; this can be overruled with the check flag. If a rule within a group has check equal to true and the rule evaluates to false, the entire group is immediately discarded and no further rules in that group are evaluated; with check equal to false, evaluation of the remaining rules continues. Note that the value of check does not affect the evaluation in groups with a single condition. Because a pattern triggers as soon as any one group is satisfied (that is, all rules in that group evaluate to true), enabling check is generally advisable to save resources since it stops further evaluation within a group once a failed rule makes success impossible under AND evaluation. However, in certain modes the check flag can be used to impose precedence or a sequential order among conditions within the same group. Consider three conditions in one group: C1 (check set to true), C2 (check set to true), and C3 (check in indifferent). Evaluation proceeds as follows: (1) C1 serves

as a prerequisite —if C1 evaluates to false, neither C2 nor C3 is evaluated—; if C1 evaluates to true, evaluation continues. (2) C2 is evaluated, if evaluated to true, evaluation proceeds to C3; otherwise C3 is not evaluated. Finally, the expression to be evaluated consists of the sensor code or the data whose value we want to evaluate, the condition operator to be used (> , <, =), and the value, either constant or by parameter, with which it is to be compared.

The five operating modes of the conditional rules are explained below:

Mode 1: Once the conditions is satisfied, the condition remains true without being reevaluated during the period of time set in the rule.

Mode 2: An expression is considered true only after it evaluates to true for the required number of consecutive cycles; any false evaluation resets the count. If a cycle evaluates to true but the required count has not yet been met, the pattern remains in a pending state and no actions for a false condition are triggered.

Mode 3: No additional restrictions apply; the condition only needs to be true in the cycle in which the pattern is evaluated.

Mode 4: Once it first evaluates as true, it isn't checked again; it will only be re-evaluated after the pattern is reset.

Mode 5: It behaves like Mode 1, except the duration is measured in cycles rather than milliseconds.

Parameter	Description
mode	Defines the evaluation mode of the rule (values 1–5).
time_cycles	Defines the duration (in milliseconds) during which all conditions must be satisfied for mode 1, Represents the number of measurement cycles available for the conditions to be fulfilled for modes 2 and 5; it is a placeholder for modes 3 and 4.
group	Identifier of the condition block to which the rule belongs. Rules in the same group are combined using logical AND.
check	If true, when a condition in this group is evaluated and not satisfied, the entire group is discarded, and the remaining expressions in that group are not evaluated.
sensor_code	Identifier of the sensor or data source (e.g., "temp", "hum").
condition	Logical operator used for evaluation (>, <, or =).
value	Reference value or threshold against which the sensor reading is compared.

Table 3: Parameters for pattern conditions configuration

In addition to the parameters required in the configuration file, the internally developed API will use other relevant variables, such as state —which stores whether the condition has been satisfied during the evaluation—; startTime —timestamp of the moment the expression was satisfied (mode 1) —; and cyclesCount and pendingCycles —the number of cycles elapsed and remaining, respectively, in modes 2 and 5—.

5.4 Illustrative Examples

In the following sub-sections, each operational mode of the MicroCEP pattern framework will be examined in detail, accompanied by practical examples that demonstrate its configuration and behaviour. For simplicity, the examples will focus on two fundamental environmental variables—temperature and humidity—which are

among the most critical parameters in greenhouse climate management. Each pattern will be illustrated through a plausible greenhouse scenario, showing how the system can detect, interpret, and respond to relevant environmental changes. The complexity of the examples will progressively increase to not only explain the operation of each mode individually but also to highlight the range and flexibility of configurations that the MicroCEP framework allows, including the use of multiple sensors, condition groups, and hybrid modes evaluation strategies:

5.4.1 Mode 1: Time-Based Mode

In this configuration, we define a time-based pattern with a time-based rule. This mode ensures that the rules are fulfilled within a set period of time. The pattern monitors the temperature sensor to detect anomalous readings above 30 °C and the humidity sensor to detect low humidity values below 10 % within a five-minute window. If the temperature rule has already evaluated to true and the humidity rule is not activated within one minute, the former is reset and must be evaluated again. The same happens but for 2 minutes when the humidity rule is activated. If both rules are satisfied, the pattern is triggered; if they are not both satisfied, the pattern is reset once the five-minute interval has elapsed.

As can be seen in Listing 1, line (1) specifies the general pattern configuration, where mode is set to 1, indicating that the pattern evaluation is governed by a fixed time window, and the time_cycles parameter is set to 300000 ms, defining a total evaluation duration of 5 minutes. Lines (2)–(3) describe the pattern condition configuration with two rules in the same group. Line (2) sets the condition to mode 1, indicating that it operates in time-based mode, and establishes a 1-minute validity window (60000 ms, via the _cycles parameter) once the condition is triggered. In this same line, the rule is associated with group 1, and the check flag is set to false, implying that the following rules in this group are evaluated regardless of whether this one is true or false. The remainder of line (2) specifies the logical expression, where the sensor identified by "temp" must report a value greater than 30. Line (3) defines the second rule, where the mode is set to 1, indicating time-based evaluation, and the time_cycles parameter is set to 120000 ms, establishing a 2-minute validity window once the condition is triggered. The rule is associated with group 1, and the check flag is set to false, which has no practical effect in this case since there are no additional rules in the group. Finally, the logical expression specifies that the sensor identified by "hum" must report a value lower than 10.

Listing 1. Example of Time-Based Configuration

```
# Pattern general configuration
(1) mode: 1, time_cycles: 300000
# Pattern condition configuration
(2)  mode: 1, time_cycles: 60000, group: 1, check: false,
sensor_code: "temp", condition: ">", value: 30
(3) mode: 1, time_cycles: 120000, group: 1, check: false,
sensor_code: "hum", condition: "<", value: 10
```

In a greenhouse scenario, this pattern configuration can be used to detect a short-lived but critical combination of extreme heat and very dry air—for example, a sudden rise in temperature around midday combined with an abrupt drop in humidity due to increased ventilation or external weather conditions. In this context, the requirement that the humidity condition must be met within one minute of the temperature threshold being exceeded ensures that only tightly correlated events are considered significant, filtering out cases where the environment briefly heats up but moisture levels remain adequate, or vice versa. When the pattern is finally triggered, it signals a potentially harmful microclimate for the crops, prompting immediate corrective actions such as activating emergency irrigation, reducing ventilation rates, deploying shading screens, or adjusting cooling systems to restore safe growing conditions.

5.4.2 Mode 2: Consecutive Cycles Mode

In this configuration, we have included two independent rules in the pattern, each rule associated with a distinct logical group. This design allows the pattern to verify multiple environmental conditions—each evaluated across consecutive measurement cycles—before declaring the event as fulfilled. Mode 2 ensures that each rule must remain true for a defined number of consecutive cycles, preventing transient fluctuations from triggering false events. When multiple groups are present, each one is evaluated independently and the pattern is satisfied when at least one group achieves its target condition set, as groups are combined disjunctively (logical OR) in the overall pattern logic. In this example, we want to detect two possible persistent situations of concern: on the one hand, in group 1 the humidity sensor ("hum") reports values below 40 % for three consecutive measurement cycles (persistent dryness); on the other, in group 2 the temperature sensor ("temp") reports values above 30 °C for two consecutive cycles (persistent overheating).

Listing 2 shows the code corresponding to this pattern. Line (1) specifies the general pattern configuration, where mode is set to 2, indicating that the pattern evaluation is governed by consecutive measurement cycles, and the time_cycles parameter is set to 1, so the rules inside have to be true *once*, as explained at the end of the paragraph. Lines (2)–(3) describe the pattern condition configuration with two independent rule groups. Line (2) sets the rule to mode 2, indicating that it also follows cycle-based evaluation semantics, and sets the time_cycles parameter to 3, requiring the condition to hold for three consecutive measurement cycles. In this same line, the rule is associated with group 1, and the check flag is set to false, implying that any subsequent rules in this group would be evaluated regardless of the outcome of this one; in this case, however, the setting has no practical effect because group 1 contains only a single rule. The remainder of line (2) specifies the logical expression, where the sensor identified by "hum" must report a value lower than 40. Line (3) defines the second rule, where the mode is again set to 2, indicating cycle-based evaluation, and the time_cycles parameter is set to 2, requiring two consecutive successful evaluations. The rule is associated with group 2, which distinguishes it from the humidity rule, and the check flag is set to false, also without practical effect since there are no additional rules in this group. Finally, the logical expression specifies that the sensor identified by "temp" must report a value greater than 30. Since the pattern time_cycles is 1, first rule has to occur only 3 consecutive cycles or the second rule 2 consecutive cycles; if the pattern

time_cycles were 2, the first rule would have to be evaluated true 6 consecutive cycles (3x2) or the second one 4 consecutive cycles (2x2).

Listing 2. Example of Consecutive Cycles Configuration

```
# Pattern general configuration
(1) mode: 2, time_cycles: 1
# Pattern condition configuration
(2) mode: 2, time_cycles: 3, group: 1, check: false,
sensor_code: "hum", condition: "<", value: 40
(3) mode: 2, time_cycles: 2, group: 2, check: false,
sensor_code: "temp", condition: ">", value: 30
```

This dual-group configuration enhances the flexibility and expressiveness of the Mode 2 pattern. In a greenhouse application, it allows the system to detect either persistent dryness or sustained overheating as separate but equally important events—without requiring both to occur simultaneously. If the humidity stays below 40 % for three consecutive cycles, the pattern raises a low-humidity alert. Alternatively, if the temperature remains above 30 °C for two consecutive cycles, the same pattern triggers a high-temperature alert. If neither condition persists for the required duration, both groups reset, and the monitoring process continues with fresh cycle counts. This configuration effectively filters out transient fluctuations due to short-term ventilation, shading, or irrigation cycles. It ensures that only stable, persistent environmental deviations initiate control actions. In practice, this avoids unnecessary system responses (such as over-watering or over-cooling), improving energy efficiency and climate stability within the greenhouse.

5.4.3 Mode 3: Single Cycle Mode

In this mode, all conditions required to detect a complex event must be satisfied simultaneously within a single iteration during pattern processing. This mode is particularly useful when the precise concurrence of multiple sensor readings defines a meaningful event, ensuring that only synchronized data points trigger the detection mechanism. For example, consider two sensors: one measuring temperature (temp) and the other measuring humidity (hum). The goal is to detect when the temperature exceeds 30 °C and, at the same time, the humidity drops below 40 % within the same monitoring cycle. Both conditions must occur simultaneously in a single iteration for the event to be recognized as valid. If either condition is not met, the pattern remains false and is re-evaluated in the next cycle.

Listing 3 shows the code corresponding to this pattern. Line (1) specifies the general pattern configuration, where mode is set to 3, indicating that the pattern evaluation is governed by single-cycle semantics in which all relevant conditions are checked within the same measurement cycle, and the time_cycles parameter is set to 0, serving as a structural placeholder that is not used in this mode but is retained for configuration uniformity across modes. Lines (2)–(3) describe the pattern condition configuration with two rules in the same group. Line (2) sets the rule to mode 3, indicating that its condition is evaluated once per cycle, and sets the time_cycles parameter to 0, again acting as an unused placeholder in this mode. The rule is

associated with group 1, and the check flag is set to true, meaning that if this rule evaluates to false, the remaining rules in the same group are not evaluated in that cycle, which avoids unnecessary processing when the first condition already fails. The remainder of line (2) specifies the logical expression, where the sensor identified by "temp" must report a value greater than 30. Line (3) defines the second rule, where the mode is again set to 3, indicating single-cycle evaluation, and the time_cycles parameter is set to 0, also used solely as a placeholder. The rule is associated with group 1, forming a conjunction with the temperature rule, and the check flag is set to false, which has no practical effect in this case since there are no additional rules after this one in the group. Finally, the logical expression specifies that the sensor identified by "hum" must report a value lower than 40. Because both rules belong to the same group, the pattern is only satisfied when both conditions are simultaneously true within the same monitoring cycle—that is, when temperature exceeds 30 °C and humidity drops below 40 % at the same instant.

Listing 3. Example of Single Cycle Configuration

```
# Pattern general configuration
(1) mode: 3, time_cycles: 0
# Pattern condition configuration
(2) mode: 3, time_cycles: 0, group: 1, check: true,
sensor_code: "temp", condition: ">", value: 30
(3) mode: 3, time_cycles: 0, group: 1, check: false,
sensor_code: "hum", condition: "<", value: 40
```

Thus, the use of Mode 3 ensures that the pattern only triggers when both sensors register their respective thresholds concurrently, thus avoiding false positives caused by asynchronous sensor readings. This mode is particularly suited to applications such as greenhouse climate management, maintaining optimal temperature and humidity levels is essential for plant health and growth. A simultaneous increase in temperature above 30 °C and a drop in humidity below 40 % within the same monitoring cycle may indicate that the ventilation system is overactive or that irrigation-based humidity regulation has failed to compensate for environmental changes. Detecting these two conditions concurrently allows the control system to identify imbalances that could lead for instance to plant water stress, or damage to delicate crops. By responding immediately to this specific co-occurrence, the system can execute targeted corrective actions such as reducing airflow, activating misting systems, or adjusting irrigation timing to restore equilibrium.

5.4.4 Mode 4: Indefinite Time Mode

In indefinite time mode, once a condition is verified, it will not be reevaluated until the pattern is reset; on the other hand, the pattern is reset once all its conditions are true. In this configuration, we also show how to combine two modes. This pattern combines two different evaluation modes to capture both short-term stability and long-term progression. The overall pattern operates under Mode 4 (Indefinite Time), meaning that there is no predefined limit for the fulfilment of its conditions. However, the first rule (humidity) uses Mode 2 (Consecutive Cycles) to ensure that a specific condition—low

humidity—remains true for several consecutive measurement cycles before the event is validated. This combination allows the system to detect persistent dryness followed by a gradual temperature increase, which together represent a realistic scenario of prolonged environmental stress.

Listing 4 shows the code corresponding to this pattern. Line (1) specifies the general pattern configuration, where mode is set to 4, indicating that the pattern evaluation remains active indefinitely until the configured conditions are satisfied, and the time_cycles parameter is set to 0, serving as a structural placeholder. Lines (2)–(3) describe the pattern condition configuration with two rules in the same group. Line (2) sets the rule to mode 2, indicating that it is evaluated based on consecutive measurement cycles, and sets the time_cycles parameter to 3, requiring the humidity condition to hold for three consecutive cycles before the rule is considered true. The rule is associated with group 1, and the check flag is set to true, meaning that if this rule does not satisfy its condition in the current evaluation, the remaining rules in the same group are not evaluated in that cycle, so the temperature rule is only considered once the humidity condition has first been stably met. The remainder of line (2) specifies the logical expression, where the sensor identified by "hum" must report a value lower than 40. Line (3) defines the second rule, where the mode is set to 4, indicating that this temperature condition is evaluated without an expiration time, and the time_cycles parameter is set to 0, again used solely as a placeholder. The rule is also associated with group 1, so it forms a conjunction with the humidity rule at the group level, and the check flag is set to false, which has no practical effect here since there are no additional rules after this one in the group. Finally, the logical expression specifies that the sensor identified by "temp" must report a value greater than 30. In operational terms, the pattern is satisfied only when humidity has remained below 40 % for three consecutive measurement cycles and, immediately after this stable dry period has been established, the temperature exceeds 30 °C; if the temperature condition is not met at that point, the humidity counter is reset in the following cycle and the combined evaluation process starts again under the same indefinite pattern context.

Listing 4. Hybrid Indefinite Time Configuration

```
# Pattern general configuration
mode: 4, time_cycles: 0
# Pattern condition configuration
(2) mode: 2, time_cycles: 3, group: 1, check: true,
sensor_code: "hum", condition: "<", value: 40
(3) mode: 4, time_cycles: 0, group: 1, check: false,
sensor_code: "temp", condition: ">", value: 30
```

This configuration defines a hybrid indefinite pattern that detects sustained dryness followed by heat accumulation within a greenhouse. The humidity condition, evaluated under Mode 2 (Consecutive Cycles), ensures that the system only recognizes dryness when it has been consistent across several readings—preventing false triggers due to transient fluctuations or sensor noise. Once this stability condition is satisfied, the temperature rule—evaluated under Mode 4 (Indefinite Time)—is evaluated indefinitely, waiting for the temperature to eventually exceed 30 °C. When both conditions are met, the system concludes that a persistent environmental imbalance has

occurred, triggering a complex event such as an overheating and drying alert. This behavior is particularly relevant in greenhouse climate management, where slow but cumulative environmental changes can stress plants and compromise crop yield. For example, a period of persistently low humidity over several measurement cycles may indicate insufficient irrigation or excessive ventilation. If this condition is later accompanied by a gradual temperature rise above 30 °C, it could signify that the cooling system is failing to compensate for ongoing water loss. Detecting this combination enables the control system to respond proactively, activating misting systems, adjusting irrigation frequency, or reducing ventilation intensity before the plants experience irreversible stress.

5.4.5 Mode 5: Cycle-Based Mode

In this configuration, the pattern operates under Mode 5 (By Cycles Mode), where the defined conditions are evaluated within a fixed number of measurement cycles rather than over continuous time. The event is triggered if all required conditions are satisfied at least once during this cycle window, even if they do not occur simultaneously or consecutively. If this limit is exceeded, the pattern is reset, avoiding indefinite monitoring

Listing 5 shows the code corresponding to this pattern. Line (1) specifies the general pattern configuration, where mode is set to 5, indicating that the pattern evaluation is governed by a fixed number of 15 measurement cycles, defining the maximum real-time duration allowed for completing the cycle-based observation window before the pattern is reset. Lines (2)–(3) describe the pattern condition configuration with two rules in the same group. Line (2) sets the rule to mode 5, ensuring that it follows the same by-cycles semantics as the pattern, and sets the time_cycles parameter to 7, establishing the observation window of seven measurement cycles in which the humidity condition is evaluated. The rule is associated with group 1, and the check flag is set to true, which ensures that this temperature is only evaluated if humidity rule is true. The remainder of line (2) specifies the logical expression, where the sensor identified by "hum" must report a value lower than 40. Line (3) defines the second rule, where the mode is again set to 5, indicating that the temperature condition is also evaluated under by-cycles semantics, and the time_cycles parameter is set to 7. The rule is associated with group 1, so both rules must be satisfied for the pattern to be fulfilled, and the check flag is set to false, with no effect. Finally, the logical expression specifies that the sensor identified by "temp" must report a value greater than 30, so the pattern is triggered when a period of low humidity within the defined cycle window is followed, within the same window, by at least one occurrence of temperature above this threshold. Please note that, for instance, if humidity rule is true in cycle 2, then there are 5 cycles left for temperature rule to be true until humidity one resets. But in the third iteration (cycle 14) both would need to be true before the pattern resets in cycle 15.

Listing 5. By Cycles Configuration (Mixed check)

```
# Pattern general configuration
(1) mode: 5, time_cycles: 15
# Pattern condition configuration
```

```
(2) mode: 5, time_cycles: 7, group: 1, check: true,
sensor_code: "hum", condition: "<", value: 40
(3) mode: 5, time_cycles: 7, group: 1, check: false,
sensor_code: "temp", condition: ">", value: 30
```

This configuration detects the co-occurrence of high temperature and low humidity within a bounded period of seven measurement cycles and a maximum duration of 15 cycles. The humidity rule ensures that the system monitors air dryness continuously in each cycle, while the temperature rule ensures that heat is only evaluated when dryness is confirmed. In a greenhouse scenario, this pattern is particularly useful for identifying periodic overheating and dryness conditions that do not necessarily persist across consecutive readings but still indicate environmental stress when they occur repeatedly within a short time frame. For example, if humidity drops below 40 % during one or more cycles and temperature exceeds 30 °C in another within 15-cycle window, the system concludes that the air–temperature balance is deteriorating. Upon detection, the greenhouse control system can automatically respond by activating misting systems, adjusting ventilation timing, or deploying shading screens to restore humidity and temperature balance.

6 Usage Methodology and Case Study

To facilitate the use of MicroCEP in any scenario, the following methodology is proposed:

Step 1. Define the input and output queues, including the IP address, port number, client identifier, and the topic to which data is sent or from which it is received.

Step 2. Establish the general configuration of the pattern. This general configuration defines the high level behaviour of the corresponding complex event patterns.

Step 3. Define the critical variables for condition monitoring, along with their respective thresholds. This involves: (a) setting the conditions associated with the complex event to be detected, (b) identifying the variables associated with the sensor types considered in each condition, and (c) establishing, for each variable, the values and parameters necessary to define the condition.

Step 4: Deploy the pattern configuration to the MCU file system.

6.1 Case Study

Figure 2 illustrates the case study setup, consisting of three ESP32-C3 MCUs: two of them connected to a DHT11 sensor and one using the board's built-in LED. The top MCU (*device1*) is equipped with a DHT11 sensor for temperature measurement, the middle MCU (*device2*) also has a DHT11 sensor used to measure humidity, and the bottom MCU (*device3*) uses the built-in LED (white light in the picture) integrated into the MCU board. The case study demonstrates the operation of a scenario in which the detection of specific ranges of temperature and humidity values in different edge devices, and their subsequent correlation by a third edge device, triggers the activation of a LED on this third device and the publication of an alert message through MQTT.

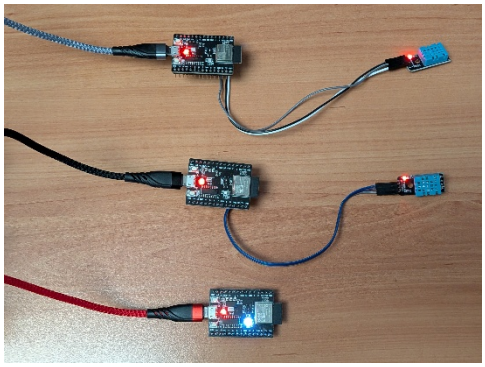

Figure 2: Microcontrollers used in the case study

The configuration files that enable the implementation of the case study are shown in Listings 6, 7 and 8, and explained in the following paragraphs.

Listing 6 shows the code for *device1*. It reads the values from the DHT11 sensor and stores the measured value in the variable *temp*. This device has a mode 3 pattern with two condition groups, both also configured in mode 3. If, during pattern evaluation, a value is detected that is too low or too high with respect to a predetermined threshold, a complex event is generated by publishing the corresponding outlier temperature value to the MQTT topic sensor/test/comb.

Listing 6. Configuration file for device 1

```
# MQTT configuration
in_mqtt_server: 192.168.1.144
in_mqtt_topic: sensor/nothing
in_mqtt_client_id: device1I
o_mqtt_server: 192.168.1.144
o_mqtt_server: 192.168.1.144
o_mqtt_topic: sensor/test/comb
o_mqtt_client_id: device1O
variableToSend: temp

# Pattern Configuration (Mode 3)
mode: 3, time_cycles: 0

# Conditions (Temperature above or below certain threshold)
mode: 3, time_cycles: 0, group: 1, check: true,
sensor_code: "temp", condition: '>', value: 44
mode: 3, time_cycles: 0, group: 2, check: true,
sensor_code: "temp", condition: '<', value: 25
```

Listing 7 shows the code for *device2*. This device operates in an analogous way, but using the values read from the DHT11 sensor and stores the measured value to the

variable *humid*. If an excessively low or high humidity value is detected, a complex event is published to the MQTT topic sensor/test/comb.

Listing 7. Configuration file for device 2

```
# MQTT configuration
in_mqtt_server: 192.168.1.144
in_mqtt_topic: sensor/nothing
in_mqtt_client_id: device2I
o_mqtt_server: 192.168.1.144
o_mqtt_topic: sensor/test/comb
o_mqtt_client_id: device2O
variableToSend: humid

# Pattern Configuration (Mode 3)
mode: 3, time_cycles: 0

# Conditions (Humidity above or below certain threshold)
mode: 3, time_cycles: 0, group: 1, check: true,
sensor_code: "humid", condition: '>', value: 70
mode: 3, time_cycles: 0, group: 2, check: true,
sensor_code: "humid", condition: '<', value: 25
```

Listing 8 shows the code for *device3*. It receives the complex events generated by device1 and device2 (containing extreme temperature and humidity values), and uses these data to detect whether combinations of temperature and/or humidity give rise to unfavourable climatic conditions. It is configured with a pattern in mode 4 that is triggered when any of the following groups is satisfied, thereby activateing the built-in LED in the third MCU and sending an alert message (*variableToSend*) through the *sensor/result* MQTT topic:

- Group 1 (conditions in mode 1). If high temperature and low humidity are detected, a complex event is generated. The check flag is set to false so that, even if the first condition is not satisfied, the second condition is still evaluated. Since both conditions are in mode 1, they have a validity period of 10 seconds (2 execution cycles of 5 seconds each).
- Group 2 (conditions in mode 5). Similar to Group 1, but with low temperature and high humidity. The check flag is again set to false for the same reason, but in this case the validity period is 2 execution cycles (with 5 seconds per cycle, this also provides a 10-second validity window, while allowing us to showcase a different mode in operation).
- Group 3 (single condition in mode 2). A single condition that must hold for 3 consecutive cycles (high temperature) to be evaluated as true.

Listing 8. Configuration file for device 3

```
# MQTT configuration
in_mqtt_server: 192.168.1.144
in_mqtt_topic: sensor/test/comb
```

```
in_mqtt_client_id: device3I
o_mqtt_server: 192.168.1.144
o_mqtt_topic: sensor/result
o_mqtt_client_id: device3O
variableToSend: The weather is unusual

# Pattern Configuration (Mode 4)
mode: 4, time_cycles: 0

# Conditions (Group 1 (Mode 1): High Temperature, Low
Humidity)
mode: 1, time_cycles: 10000, group: 1, check: false,
sensor_code: "temp", condition: '>', value: 44
mode: 1, time_cycles: 10000, group: 1, check: false,
sensor_code: "humid", condition: '<', value: 25

# Conditions (Group 2 (Mode 5): High Humidity, Low
Temperature)
mode: 5, time_cycles: 2, group: 2, check: false,
sensor_code: "temp", condition: '<', value: 25
mode: 5, time_cycles: 2, group: 2, check: false,
sensor_code: "humid", condition: '>', value: 70

# Conditions (Group 3 (Mode 2): Continous High Temperature)
mode: 2, time_cycles: 3, group: 3, check: true,
sensor_code: "temp", condition: '>', value: 44
```

It is readily visible from the pattern code that every configuration file consists of three main blocks as per steps 1-3 of the usage methodology: The first block defines the input and output queues (MQTT IP details and IDs, as well as the overall event variable) (step 1); the second block defines the overall pattern behaviour (step 2) and the final block contains the conditions which are expected to occur within the given time/cycle interval (step 3). Note that the input queue in the first and second MCU are set to sensor/nothing since the values are directly read from the sensor itself.

Overall, the use case illustrates the simplicity of using the framework and the fundamental feasibility of the approach. The full source code for this example is available in the repositories referenced in footnote 1.

7 Evaluation

For the experiments we have used three ESP32-C3, and simulated temperature and a humidity sensor readings. A Mosquitto broker running on the local network in a PC has been used for all the experiments. We have developed throughput tests, with one and multiple rules in the pattern, first with increasing input and output data in the brokers and afterward with increasing input data in the brokers but with a limited number of messages at the output. Then we have proceeded to perform tests, removing the input/output overhead by simulating data internally in the MCU to assess the

throughput of the pattern evaluation in isolation. Finally, we have performed an additional evaluation of execution times, communication latency between devices and heap memory usage in a more realistic scenario with several MCUs interacting. Note that all the results obtained in the tests are also available in the repository previously referenced.

7.1 Single-MCU throughput tests with increasing input/output data volume

Throughput tests with one MCU and increasing input and output data volume have been performed with the aim of assessing the MQTT message reception capacity of the MicroCEP nodes, identifying the maximum number of messages they are able to receive and process. The pattern was configured in mode 3, guaranteeing that neither the pattern state nor the condition state remains true across different execution cycles of the program, and its condition is always satisfied, and therefore an output message is always published to the output queue. To deliberately overload the MCU, the waiting interval between two consecutive evaluations of the pattern was set to 0 ms; that is, there is no delay between cycles. Each experiment with a variable number of variable number of messages sent per second to a MicroCEP node lasts 60 seconds.

As shown in Table 4, the MicroCEP nodes provide reliable reception of up to approximately 250 messages per minute (4 messages per second) before data losses begin to occur. When messages arrive at higher rates, they can be overwritten by later messages before the MQTT client in the node has time to process them (as can be observed for input rates of 6 and 10 messages per second). This bottleneck is likely caused by limitations in the library used for queue management, which will be investigated in future versions of MicroCEP.

Input/Output rate (events per second)	**1**	**2**	**4**	**6**	**10**
Received (events)	60	120	240	248	260
Expected (events)	60	120	240	360	600
Reliability (percentage)	100,00	100,00	100,00	68,89	43,33

Table 4: Results of throughput tests with increasing input/output data volume

Once this upper threshold of messages manageable by the node was identified, the experiment was repeated with a permanent input and output rate of 4 events per second respectively and an increasing number of conditions in the pattern, from 1 to 20, again defined so that the conditions are always satisfied and ensuring that all condition groups are evaluated. The results showed that the MCU Kept a 100% reliability processing all the messages received regardless of the number of conditions in the pattern.

7.2 Single-MCU throughput with increasing input and limited output data volume

Previous experiments were repeated, but in this case the pattern is configured to be satisfied only once per second, so that at most one output message is published per second. The objective of this experiment is to reduce the number of published messages to clearly determine whether the bottleneck lies in the publication of results or in the reception of input data. Apart from the differences described above, all other parameters are identical to those in Experiment 1, including the number of iterations performed.

The results, as can be observed in Table 5, show that the limit of messages that can be received and processed is again reached at around 4 messages per second. For 6 and 10 messages per second, a non-negligible fraction of messages is lost, although the loss is slightly lower than in the previous experiment. This confirms that the bottleneck is in message reception rather than in message publication.

Input rate (events per second)	**1**	**2**	**4**	**6**	**10**
Received (events)	60	60	60	49	20
Expected (events)	60	60	60	60	60
Fiability Rate (percentage)	100,00	100,00	100,00	81,67	33,33

Table 5: Results of throughput tests with increasing input data volume

On the other hand, when varying the number of conditions from 1 to 20 at the previously established threshold rate of 4 events per second, the pattern is detected in all expected cases, keeping the reliability to 100%. Thus, the number of evaluated conditions remains a factor with no impact on performance.

7.3 Isolated pattern evaluation throughput with simulated data

An additional experiment was performed to determine how many times the complete execution logic of a pattern can be run when all operations related to the publication of the resulting MQTT message are removed. For this purpose, a specific sensor code was introduced in the program, which always returns a fixed value greater than 0. Using this sensor code, a configuration file was defined that specifies a pattern with a single condition that is intentionally never satisfied. This prevents the publication of result messages. The waiting time between execution cycles was set to 0 ms so that a new cycle begins immediately after the previous one finishes. The experiment duration was 1 minute. The results showed that the system reached an average time of 1.004 ms for reading the sensor value from the memory and executing the pattern.

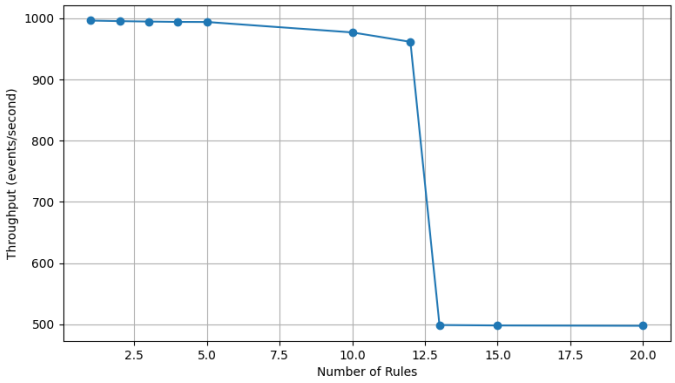

Figure 3: Throughput with a varying number of conditions in the pattern

When varying the number of conditions from 1 to 20, we observed, as shown in Figure 3, that the throughput remains approximately stable up to 12 conditions; however, at 13 conditions the performance drops sharply. A priori, this is not expected to have a major impact, since it is unlikely that so many conditions will need to be evaluated simultaneously on a single microcontroller. In any case, we will investigate the causes of this behavior and try to improve it in future versions of the code.

7.4 Evaluation in a multi-MCU collaborative scenario

A final experiment was conducted to demonstrate how the system performs under a realistic collaboration scenario, particularly the one with three devices described in the case study section. To streamline the experiments the real sensor readings and the actuation on the led have been replaced by simulated temperature (device1) and humidity (device 2) readings, and alert message published to a queue (device 3); all the messages involved in this experiment passed through a single MQTT broker. The input publication rate for MCU 1 and 2 was set to half of the threshold detected in previous experiments, therefore both devices receive two sensor values, randomly emulated within a predefined interval, per second. The waiting interval between program execution cycles was set to 5 seconds for the three MCUs, in order to demonstrate the feasibility of the system under normal operating conditions. We measured (i) pattern execution time, (ii) execution time of individual expressions, (iii) communication latency between devices, and (iv) memory usage. In the following lines, we summarise the results obtained.

(i) The pattern execution time remains low, between 0.7 and 1.3 ms on all three devices when the pattern is not detected, with device3 being slightly slower due to its higher number of conditions. In cycles where the pattern is detected, as shown in Figure 4, the execution time increases to around 2 ms because a publication, included in the measured time, is performed.

(ii) The execution times of individual expressions, as shown in Figure 5, are low and relatively stable. This is explained by code optimizations: certain expression evaluations are skipped when they cannot contribute to satisfying the pattern (e.g., when the pattern is already satisfied, the condition remains within its validity window, or the required data are not present in memory). In these cases, expressions do not need to search and read the corresponding sensor data in memory, resulting in significantly shorter execution times than in cases where such a search is required. Moreover, comparing pattern execution times with the sum of expression execution times shows that, in cycles where a message is published, the pattern execution time is substantially higher than the cumulative expression time, reinforcing the observation that message publication is comparatively expensive.

(iii) The latencies between devices (i.e., the elapsed time from when a message is published by device1 or device2 until it is received by device3) are low, typically between 40 and 100 ms, with a few outliers around 200 ms attributable to Wi-Fi.

(iv) Heap memory usage remains stable over the 20 cycles for all three devices, at around 30% (approximately 90 KB of heap memory).

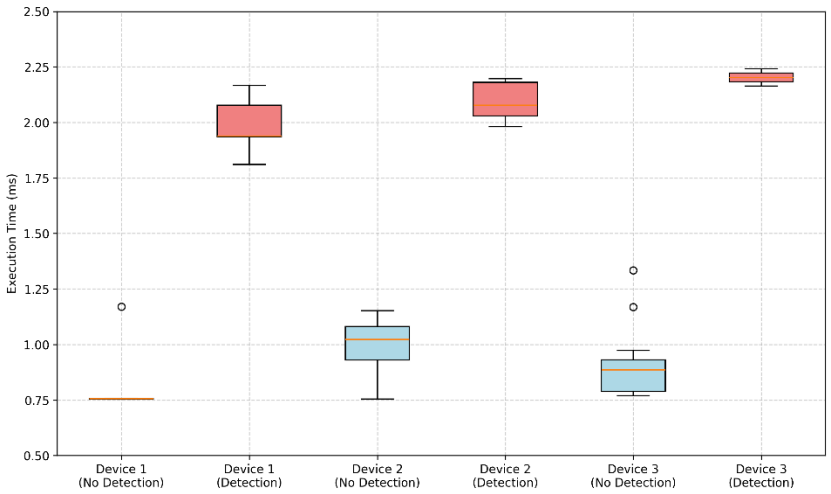

Figure 4: Pattern execution time depending on whether the condition is met or not

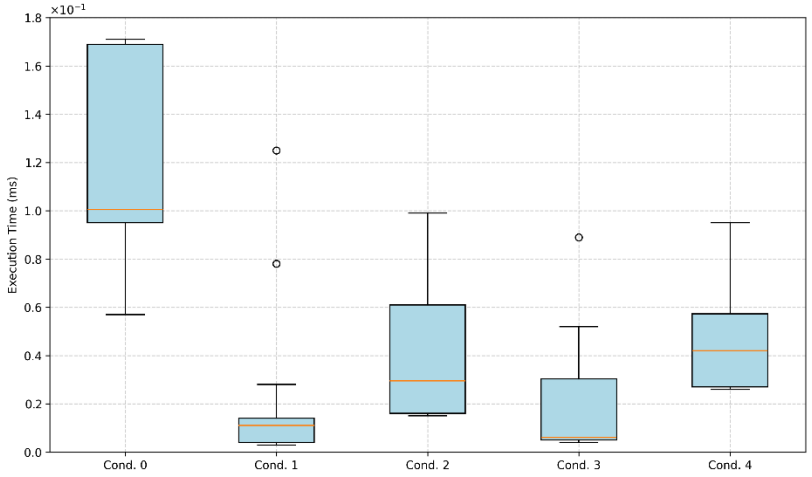

Figure 5: Isolated conditions execution time

8 Discussion

MicroCEP provides a variety of functionalities to facilitate real-time pattern detection based on data sensed by one or more MCUs and/or data received via messaging. Undoubtedly, this could be achieved without MicroCEP, but our contribution simplifies this process by providing a framework, supported by an implementation and usage methodology that allows for the generalization of pattern design, regardless of the application domain. Moreover, the system also enables thresholds used in comparisons

or correlations to be updated at runtime, without the need to halt execution, modify the code, and reload it onto the MCU.

This ability to easily program and detect patterns in real time, based on data sensed by the MCU and data from other MCUs or additional sources in the environment, allows to create a network of MCUs that can exchange information on demand. Their decisions are then based not only on their local data but also on data and/or situations detected in other MCUs within the environment, thus facilitating decision-making within the IoT device itself (i.e. right next to the sensor).

From the experiments, we can conclude that message reception can become a potential bottleneck when the incoming rate is sufficiently high, whereas the number of conditions handled in each pattern does not have a notable impact on the number of messages that the system can process when sending four messages per second. However, when we stress the system to obtain the maximum pattern throughput, with neither input nor output queues, performance clearly degrades from 13 conditions onwards. Finally, the experiment in a more realistic scenario shows that the platform can execute collaborative patterns with low execution times, low communication latencies and stable memory consumption. This not only demonstrates the low resource consumption and optimisation of the code, but also indicates that additional sensor codes could be stored in memory without risking exhaustion of heap space.

Finally, to better position our proposal with respect to existing lightweight CEP engines, Table 6 summarizes the main similarities and differences between MicroCEP, and µCEP, which are the only approaches we could find really developed for MCUs. It shows that both solutions contribute to the field of CEP on MCUs, each providing features that the other lacks. The main drawbacks of µCEP are that it is not open source and that it does not include experimental scenarios in which multiple MCUs collaborate. Conversely, although microCEP exhibits slightly lower performance, it shows better scalability than µCEP as the number of conditions increases.

Approach	MicroCEP (this work)	µCEP / TinyML [Ren et al., 2021]
Target platform	Low-cost RISC-V MCUs (ESP32-C3, 400KB de RAM y 4M flash).	Arduino Nano 33 BLE (Cortex-M4, 256 KB RAM, 1 MB flash).
Core technology	MicroCEP engine in C++; configuration via REST API; event exchange via MQTT.	µCEP engine in C/C++; TinyOL/TinyML for inference and learning.
Rule / pattern language	Custom pattern language in config file; time-based and cycle-based windows, comparative operators, concurrent/sequential.	ETALIS-derived language; and, or, seq, nseq, kseq, lambda (avg, sum, min, max, abs), count, range; simple windows.
Runtime update	Thresholds updatable at runtime via REST; rules changed offline.	CEP rules injected at runtime.
Collaboration	Network of MCUs exchanging simple/complex events (MQTT).	Local sensor exchanging simple events data.
Execution time	average time of 1.004 ms for reading the sensor value from the memory and executing the pattern.	Average time of 0,68 ms for executing a pattern.
Throughput	Clear drop in throughput when the number of conditions exceeds 12.	Clear drop in throughput when the number of conditions exceeds 5.
Open source	Yes	No

Main limitations	One pattern per MCU; single input/output topic/queue.	Limited semantics and temporal expressiveness; performance measured on internal loop, no evaluation of collaboration scenarios

Table 6: Comparison between MicroCEP and µCEP [Ren et al., 2021]

On the other hand, MicroCEP has several limitations that we will be dealt with in the future. Firstly, currently the system only supports a single configuration file; that is, we can only define a unique pattern per MCU (although the pattern, as we have seen, can include various conditions and be quite complex). This is undoubtedly the most significant limitation, but the implementation of this system has served as a proof of concept to successfully assess the viability of our proposal. We are currently working on extending the configuration file specification to support the definition of multiple independent patterns that can be executed on the same MCU. We will continue to rely on a single configuration file; however, the user will now be able to declare as many patterns as needed (through the new @pattern label), each with its own set of conditions organized into groups and following the same logic as in the current version. Secondly, the reception of simple events and the submission of complex events are currently limited to a single messaging queue. Thus, we are also working to enable the use of multiple queues or topics for both input and output messages, allowing the association of each pattern with its own topic and message payload to be published, thereby enabling the system to send the relevant information of each pattern to different topics as needed. At run time, the system will evaluate the conditions of the different patterns independently and will publish the message corresponding to each pattern that has been detected in the execution cycle.

9 Conclusion and Future Work

MicroCEP provides a set of mechanisms that support real-time pattern detection over data sensed by one or more MCUs and over data received through messaging. Built as a generic framework, with a reference implementation and a usage methodology, MicroCEP enables the definition of CEP patterns independently of the application domain.. Furthermore, the system allows thresholds used in comparisons and correlations to be updated at runtime without stopping execution, modifying the code, or reloading it onto the MCU. This capability to easily program and detect patterns in real time, based on both locally sensed data and information received from other MCUs and external sources, facilitates building networks of collaborating MCUs. Decisions can thus be taken not only from local measurements but also from situations detected in neighbouring devices, bringing decision-making as close as possible to the sensor.

In our future work, in addition to addressing the mentioned limitations, we aim to introduce the capability of fully updating patterns at runtime (not only thresholds), which would represent a substantial improvement to the proposal; enable hosting the message broker in the MCU to reduce dependencies on external systems; extend the set of operators that can be used in the patterns; improve response times and conduct additional testing to evaluate the response speed of the framework for the different operators in the patterns.

Acknowledgments

This work was supported in part by the I+D+i grant PID2021-122215NB-C33 (AwESOMe) funded by MICIU/AEI/10.13039/501100011033 and by ERDF/EU. A. García-de Prado also acknowledges that this work was partially supported by the ERDF Andalusia Operational Programme 2021–2027 and by the Andalusian Regional Government's Ministry of University, Research and Innovation under project PICOC [FEDER-UCA-2024-A1-21].

References

[Apache, 2025] Apache: 'Drools'; (2025). Retrieved 23 November 2025, from https://github.com/apache/incubator-kie-drools

[Bazan-Muñoz et al., 2024] Bazan-Muñoz, A., Ortiz, G., Augusto, J. C., Garcia-de-Prado, A.: 'Taxonomy and software architecture for real-time context-aware collaborative smart environments'; Internet of Things, 26 (2024), 101160. https://doi.org/10.1016/j.iot.2024.101160

[Choochotkaew et al., 2017] Choochotkaew, S., Yamaguchi, H., Higashino, T., Shibuya, M., Hasegawa, T.: 'EdgeCEP: Fully-Distributed Complex Event Processing on IoT Edges'; In 2017 13th International Conference on Distributed Computing in Sensor Systems (DCOSS). Ottawa, ON: IEEE (2017), 121–129. https://doi.org/10.1109/DCOSS.2017.14

[EsperTech, 2025] EsperTech: 'Esper'; (2025). Retrieved 23 November 2025, from http://www.espertech.com/esper/

[Espressif, 2025] Espressif: 'ESP32-C3 Series'; (2025). Retrieved 23 November 2025, from https://www.espressif.com/en/products/socs/esp32-c3

[Kharboucha et al., 2018] Kharboucha, A., El Khoukhi, H., Nait Malek, Y., Bakhouya, M., De Florioa, V., El Ouadghiri, D., et al.: 'Towards an IoT and Big Data Analytics Platform for the Definition of Diabetes Telecare Services'; SSRN Electronic Journal (2018). https://doi.org/10.2139/ssrn.3186346

[Kimovski et al., 2021] Kimovski, D., Matha, R., Hammer, J., Mehran, N., Hellwagner, H., Prodan, R.: 'Cloud, Fog, or Edge: Where to Compute?'; IEEE Internet Computing, 25, 4 (2021), 30–36. https://doi.org/10.1109/MIC.2021.3050613

[KrishnA et al., 2024] KrishnA, M. H., Manjunatha, Kumar, R., Singh, N., Kumar, A., Ftaiet, A. A.: 'Complex Event Processing in Web Streams with Ontology-Based Abstraction Layers for Smart City Frameworks'; In 2024 International Conference on Trends in Quantum Computing and Emerging Business Technologies. Pune, India: IEEE (2024), 1–6. https://doi.org/10.1109/TQCEBT59414.2024.10545107

[Lan et al., 2019] Lan, L., Shi, R., Wang, B., Zhang, L., Jiang, N.: 'A Universal Complex Event Processing Mechanism Based on Edge Computing for Internet of Things Real-Time Monitoring'; IEEE Access, 7 (2019), 101865–101878. https://doi.org/10.1109/ACCESS.2019.2930313

[Ma et al., 2015] Ma, M., Wang, P., Chu, C.-H., Liu, L.: 'Efficient Multipattern Event Processing Over High-Speed Train Data Streams'; IEEE Internet of Things Journal, 2, 4 (2015), 295–309. https://doi.org/10.1109/JIOT.2014.2387883

[Ortiz et al., 2022] Ortiz, G., Castillo, I., Garcia-de-Prado, A., Boubeta-Puig, J.: 'Evaluating a Flow-Based Programming Approach as an Alternative for Developing CEP Applications in IoT'; IEEE Internet of Things Journal, 9, 13 (2022), 11489–11499. https://doi.org/10.1109/JIOT.2021.3130498

[Pietrzak et al., 2012] Pietrzak, P., Lindgren, P., Makitaavola, H.: 'Towards a lightweight CEP engine for embedded systems'; In IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society. Montreal, QC, Canada: IEEE (2012), 5805–5810. https://doi.org/10.1109/IECON.2012.6389134

[Ren et al., 2021] Ren, H., Anicic, D., Runkler, T. A.: 'The synergy of complex event processing and tiny machine learning in industrial IoT'; In Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems. Virtual Event Italy: ACM (2021), 126–135. https://doi.org/10.1145/3465480.3466928

[Ren et al., 2022] Ren, H., Anicic, D., Runkler, T. A.: 'Towards Semantic Management of On-Device Applications in Industrial IoT'; ACM Transactions on Internet Technology, 22, 4 (2022), 1–30. https://doi.org/10.1145/3510820

[Rosa-Bilbao et al., 2023] Rosa-Bilbao, J., Boubeta-Puig, J., Rutle, A.: 'CEPEDALoCo: An event-driven architecture for integrating complex event processing and blockchain through low-code'; Internet of Things, 22 (2023), 100802. https://doi.org/10.1016/j.iot.2023.100802

[Sudharsan et al., 2022a] Sudharsan, B., Breslin, J. G., Ali, M. I.: 'ML-MCU: A Framework to Train ML Classifiers on MCU-Based IoT Edge Devices'; IEEE Internet of Things Journal, 9, 16 (2022a), 15007–15017. https://doi.org/10.1109/JIOT.2021.3098166

[Sudharsan et al., 2022b] Sudharsan, B., Breslin, J. G., Tahir, M., Intizar Ali, M., Rana, O., Dustdar, S., Ranjan, R.: 'OTA-TinyML: Over the Air Deployment of TinyML Models and Execution on IoT Devices'; IEEE Internet Computing, 26, 3 (2022b), 69–78. https://doi.org/10.1109/MIC.2021.3133552

[WSO2, 2019] WSO2: 'Siddhi'; (2019). Retrieved 23 November 2025, from http://siddhi.io/

[Zhang and Tao, 2021] Zhang, J., Tao, D.: 'Empowering Things With Intelligence: A Survey of the Progress, Challenges, and Opportunities in Artificial Intelligence of Things'; IEEE Internet of Things Journal, 8, 10 (2021), 7789–7817. https://doi.org/10.1109/JIOT.2020.3039359