Runtime Deployment, Management and Monitoring of Web of Things Systems

Ege Korkan[1], Miguel Romero Karam[1], Sebastian Kaebisch[2], Sebastian Steinhorst[1]
[1] *Technical University of Munich,* Germany, Email: {ege.korkan, miguel.romero, sebastian.steinhorst}@tum.de
[2] *Siemens AG,* Germany, Email: {sebastian.kaebisch}@siemens.com

***Abstract*—Internet of Things (IoT) applications have been traditionally programmed using predefined device-level frameworks, tightly coupling software with the underlying hardware. The Web of Things (WoT) on the other hand abstracts interfacing with devices through the WoT Thing Description (TD) standard, allowing to program applications for Systems of Things, referred to as Mashups. In addition to the benefit of being programmed in high-level languages, WoT Mashups can be ported into serialization formats such as the WoT System Description (SD) for better insight and verification of the Mashup. Although WoT readily facilitates the development of WoT Mashups, it lacks a sound mechanism for remote deployment, management, and monitoring of such. In this paper, we propose a method and its corresponding open-source implementation, the WoT Runtime Framework, to close the development cycle of WoT Mashups. It allows users to deploy WoT Mashups either as code or in System Description format, manage their life cycle, verify the correct functionality and monitor both runtime and Mashup-specific information. The evaluation proves inter-runtime communication between multiple instances of the WoT Runtime is possible, and demonstrates this with examples from the industrial automation and smart agriculture domains.**

***Index Terms*—Web of Things, Internet of Things, WoT Runtime, Remote Deployment**

I. Introduction

The Internet of Things (IoT) has become indispensable across industries and verticals, but exponential growth and lack of standards lead to an ever more fragmented IoT ecosystem. Furthermore, the inherent complexity and interdisciplinary nature of the IoT environments makes its implementation challenging for adopters. Despite an increased offering of off-the-shelf devices to alleviate the complexity from manufacturing through deployment, value-generation only truly begins after devices become operational. At this stage, users are left responsible for harvesting value from their implementations and the lack of interoperability and high-coupling between hardware and software significantly hinders this process.

The Web of Things (WoT) standards by the World Wide Web Consortium (W3C) are intended to enable interoperability across IoT platforms and application domains [1]. The W3C WoT proposes the Thing Description (TD) [2] as the central component to uniformly describe the interaction interfaces of IoT devices or Things, as referred to in this paper. Analogous to the index.html of web-pages, a TD acts as the single point of entry for interfacing with Things.

Motivation: An open and community-driven WoT has the potential to bridge the gap between the device and application domains. The WoT concerns itself with the application layer, above the inherent complexities present in the lower-level IoT domain. By doing so, it enables programming IoT applications at a higher-level, serving as a unified abstraction for interconnecting Things and Systems of Things, or Mashups. As an extension to the TD, the WoT System Description (SD) [3] introduces a means of specifying this application logic, bringing composability and serialization to Mashups, as the TD does for Things.

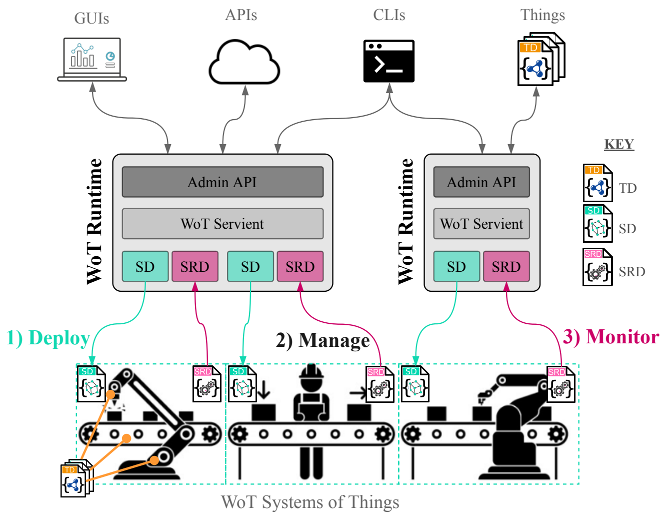


Fig. 1: Overview of the proposed WoT Runtime Framework with two running instances, each of which can be remotely administered by any number of clients through the admin API, allowing to remotely deploy WoT System Descriptions (SDs) (1), manage their lifecycle (2) and monitor them via the WoT System Runtime Description (SRD) (3).

Contributions: While the SD facilitates composition and automatic code generation for the deployment into WoT runtimes, doing so is inconvenient for developers in the current WoT ecosystem. This paper leverages the SD and provides the missing block in the WoT development cycle: a systematic approach for remote deployment, management, and monitoring of WoT Things and Systems applicable to new and existing IoT solutions whose overview can be seen in Fig. 1. Here, one can see individual Things represented by TDs and are part of Mashups represented by SDs.

Parting from the assumption of pre-deployed Things, we introduce a method and its corresponding open-source implementation, the WoT Runtime Framework, to provide:

- remote deployment, management, and monitoring of WoT Mashups into safe sandboxed runtimes, shown by the numbered arrows in Fig. 1.

- a systematic approach for control and visibility of WoT system runtimes via a runtime controller in form of a TD-compliant WoT System Runtime Description (SRD), allowing to compose multi-layered architectures,
- observability and verification of WoT Systems via runtime logs, metrics and traces.

Section II further introduces relevant WoT aspects, while Section III and IV go into methodology and implementation details respectively. Aforementioned contributions are evaluated in Section V with smart agriculture and industrial automation case studies. Related work is discussed in Section VI and Section VII concludes.

II. Web of Things

The W3C WoT aims to facilitate usability of the IoT and its interoperability by exposing devices to applications as WoT Things, decoupling them from the details of their underlying protocols and data models. Despite being first conceptualized in 2009 [4], the first official standards, the WoT Architecture [1] and the WoT Thing Description [2], were published by the W3C WoT Working Group in April 2019. The central concept in the WoT is that Things expose their Interaction Affordances and describe them through a TD document, which can then be consumed by other Things or services for interaction. Interaction Affordances can be Properties, Actions or Events and provide a means of modeling the network-facing interface of physical or virtual Things, serializable through a TD document.

A. *WoT Thing Description*

A Thing Description provides a data format for describing metadata and network-facing interfaces of Things. In the WoT context, a Thing is an abstraction of a physical or virtual entity that provides interactions to and participates in the Web of Things [2]. A TD instance is composed of four main components: Thing Metadata, a set of Interaction Affordances to describe interactions, Data Schemas for machine-readability of exchanged data and Web Links to express relationships to other Things or documents on the Web. TDs are encoded in JSON-LD[1] by default and instances can be hosted by the Thing itself or externally, allowing both resource-restricted and legacy devices to take part in the WoT.

B. *WoT Scripting API*

The Scripting API is an optional building block in the W3C WoT and provides a convenient way to extend WoT capabilities and implement WoT applications [5]. Scripting requires the ability to run a WoT Runtime and script management, for which gateways or browsers lend themselves well and are thus commonly used. The WoT Scripting API [5] defines an application programming interface (API) to allow scripts to discover and operate Things and expose locally defined Things. The WoT Interface represented by the Scripting API strictly follows the WoT Thing Description specification and provides layered interoperability based on how Things are discovered and used: *exposed* and *consumed*.

[1]https://www.w3.org/TR/json-ld11

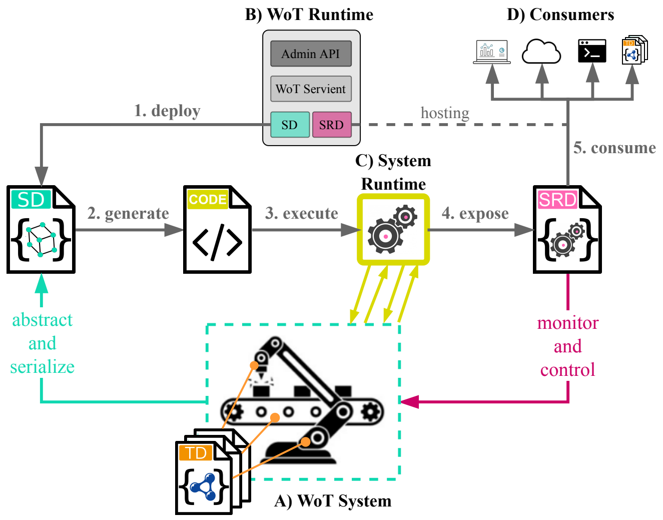


Fig. 2: Overview of a WoT System (A) abstracted and serialized as an SD in an instance of the WoT Runtime (B) running in a remote host. Through the admin API (application programming interface) of the WoT Runtime, deployment (1) of the SD is made possible, which triggers the code-generation (2) and execution (3) steps. This results in a running WoT System Runtime process (C), which is automatically hosted by the WoT Runtime (B) and exposed (4) as an SRD for clients to consume (5) and thus interact with it, enabling its monitoring and control.

Consuming a Thing: Creates a local programmatic object which exposes the described WoT Interactions. A Consumed Thing allows the TD to be introspected and to read, write and observe Properties, invoke Actions and subscribe to Events of the corresponding Thing.

Exposing a Thing: Creates a Thing to be exposed on the network based on its TD by generating the corresponding protocol bindings. An Exposed Thing allows adding, removing and registering service handlers for Properties, Actions and Events and emitting Events.

C. *WoT System Description*

The WoT System Description introduced in [3] enhances the TD with additional keywords for the description of WoT Systems, or Mashups. By building on the TD format, the SD shares its advantages to provide a textual format for describing Systems of Things. To do so, the SD introduces a means to specify the execution of interactions and represent application logic consisting of programming structures e.g. `if`-statements, `for`-loops and `wait` commands.

III. Methodology

In this section, we present our systematic approach for remote deployment of WoT Systems into individual sandboxed runtime environments as well as the methodology used for management of those runtimes and monitoring of Mashup-specific information. We first establish the requirements of this paper, then introduce our approach.

A. Requirements

The following requirements set the focus and outline the goals of our WoT Runtime methodology, and drive our approach, explained in Section B:

- **R1:** Remote deployment, management, and monitoring of runtime instances from arbitrary clients.
- **R2:** Exposure over the network to enable consumption of WoT Runtimes as if they were Things, allowing multi-layered architectures.
- **R3:** Monitoring and verification ability for obtaining runtime and Mashup-specific information.
- **R4:** Modular WoT Runtime architecture to allow composing robust, loosely-coupled software systems.
- **R5:** A reference user interface (UI) client to visually exhibit core features.

B. Approach

Our approach is structured in five main steps: deploy, generate, execute, expose, and consume, as illustrated in Fig. 2 and further detailed below with matching numbers.

1. Remote Deployment: Deploying WoT Systems into secure runtime environments as code or in SD format represents a core of our methodology. While the SD already provides a serialization format for defining Mashups, it lacks a means to deploy these in an automatic and reproducible way. To do so, the core module of the WoT Runtime Framework exposes a uniform network interface to allow for both programmatic and over-the-air deployments. By detaching the deployment target (the core module) to the deployment source (any client capable of communicating with the admin API), the framework remains decoupled and thus flexible. Borrowing from well-established REST principles, our approach enables composition of distributed architectures, where WoT Runtimes become another component of the overarching system.

2. Code Generation, Transpilation and Adaptation: Considering how WoT Systems should be deployable in SD format, a means of generating executable code is necessary. Our methodology extends on the open-sourced algorithm introduced by the WoT System Description [3] to generate and transpile SDs into source code for the WoT Runtime to execute programmatically. At this stage, the parseable SD logic permits generating highly-optimized code for efficient execution and alignment to WoT Scripting API standard.

In addition to the code generation step and transpilation steps, a code adaptation step is necessary to permit tracking code execution without polluting the auto-generated WoT System code, neither syntactically nor in its execution. This way, WoT System logic remains true to the originally deployed version. We therefore propose static code analysis and insertion of asynchronous annotations as wrapper functions, or hooks, to enable monitoring code execution at runtime. Insertion of these asynchronous hooks remains non-blocking, maintaining the efficiency of code execution.

3. Execution Control and Environment: Next to secure deployment of WoT Systems to remote hosts, the safety of the WoT System Runtime environment itself is of high priority. We propose sandboxing the WoT System Runtime in a secure, self-contained execution context. This prevents

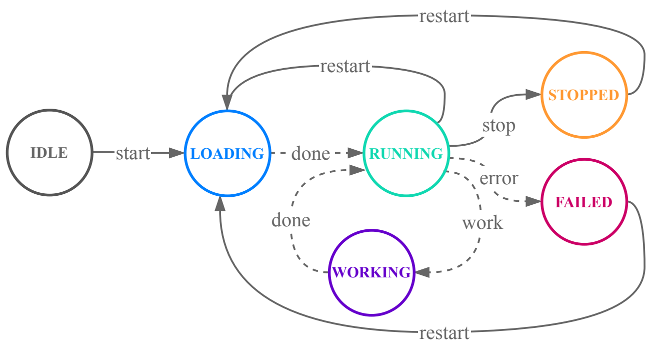


Fig. 3: State machine illustrating the possible operational states and transitions for running WoT System Runtimes. Deterministic aspects of the state machine ensure correct states and transitions at all times, to allow for a single-source of remote truth between the actual running runtime and its state, managed by the core module and consumable by clients. The state machine is directly mapped into the SRD, which exposes the current active state via the `status` Property and maps transitions as Events which fire on state change.

code from escaping its execution context into the host, which has the capability of running multiple execution contexts simultaneously. The sandbox runs its own isolated process, side-by-side to the main process of the core module and can also require permitted external libraries and built-in modules. Each process can be monitored and controlled remotely while an internal control flow mechanism guarantees state and state transitions of the WoT System Runtime remain deterministic, as shown in Fig. 3.

4. WoT System Runtime Description (SRD): We propose the TD-compliant SRD as a means for standardizing runtime management and both runtime and Mashup-specific monitoring of running WoT Systems. The SRD is dynamically generated and automatically exposed over the network by the WoT Runtime for each running WoT System Runtime instance. The SRD exposes WoT System Runtime information such as `status` and `memoryUsage` as Properties, `start`, `restart`, and `stop` commands as Actions and state changes or transitions as Events, such as `start`, `work`, and `error`. Mapping the entire operational state chart of the WoT System Runtime shown in Fig. 3 into corresponding Interaction Affordances in the SRD enables real-time synchronisation between the runtime and its digital representation, resulting in precise verification of operational safety.

For the actual monitoring and verification of WoT System Runtime instances, we propose an approach that assimilates that of conventional software, borrowing from standard observability patterns, namely logs, metrics and traces. Aligning to well-know standards allows monitoring of WoT Runtimes via existing software monitoring methods and tools, such as OpenTelemetry[2]. This further reduces implementation effort and promotes monitoring of WoT Runtimes with a systematic, out-of-the-box approach based on the three pillars of observability [6]:

- **Logs:** an immutable, timestamped record of discrete events over time

[2]https://opentelemetry.io/

- **Metrics:** a numeric representation of data over time
- **Traces:** a series of events that encode the end-to-end request flow through a system

5. Consuming the SRD: By fully complying to the TD specification, the SRD permits its consumption in the WoT context as if it were a Thing. This enables composition of multi-layered WoT applications in which WoT System Runtimes are available for interaction, as shown in Fig. 1. Concretely, consuming SRDs permits clients to remotely manage and monitor running instances of the WoT Runtime. Moreover, the serializable SRD format facilitates automatic graphical user interface (GUI) generation for visual interaction in the form of visuals and controls. Thus, the SRD describes a flexible yet capable interface for arbitrary client consumption, such as within WoT applications or from GUI clients. The SRD is automatically hosted by the WoT Runtime for each deployed and running WoT System.

IV. Implementation

Our approach, which is detailed in Section III.B, is delivered in form of a modular and extensible software framework, with the functional source code publicly available[3], ready for use and eventual extension. We first explain its architectural components in Section A, followed by the details on how our implementation sets the execution environment and controls the execution, in Section B and C, respectively.

A. WoT Runtime Framework

Here, we present an architectural overview of the modular software framework proposed by this paper: the WoT Runtime with its core and UI modules.

Core Module: The entire framework is structured around the standalone core module, which by itself should provide all intended functionality (remote deployment, management, and monitoring) and a portable distribution to allow for decentralized deployments both to the edge and cloud. It provides all the main features in a single package and enforces loose coupling through its RESTful API to promote development of third-party clients to directly interact with it, rather than locking users into specific clients. This allows integrating any number of optional clients, such as GUIs or command-line interfaces (CLIs). The Core Module strictly aligns to the requirements presented in Section III.B.

UI Module: In addition to the core package, we provide a multi-tenant GUI which exposes the entire set of features from the core package as a visual interface to further streamline the development cycle of WoT Systems. It consists of a web application to allow targeting multiple WoT Runtime processes running across any number of hosts, both at the cloud or edge of the network, as long as they are accessible through it. The single requirement being that the IP address and port of the host is known and accessible for the GUI to connect to.

B. Execution Environment

Sandboxing: The WoT Runtime Framework sandboxes each running WoT System Runtime in its own sandboxed

[3]https://github.com/tum-esi/wot-runtime

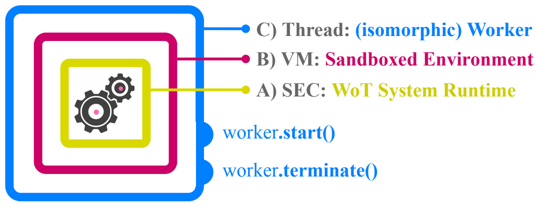

C) Thread: (isomorphic) Worker
B) VM: Sandboxed Environment
A) SEC: WoT System Runtime
worker.start()
worker.terminate()

Fig. 4: The sandboxing approach implemented by the WoT Runtime Framework to encapsulate self-contained script execution contexts (SEC) of each WoT System Runtimes (A) in a sandboxed environment (B) using virtual machines (VMs) and consequently in an optionally isomorphic worker (C), enabling execution control of the core process by directly controlling the outermost worker instance, which conveniently exposes a `start()` and `terminate()` methods.

execution contexts by the means of the `vm2` library[4]. `vm2` is a sandboxing utility to allow execution of untrusted code, protecting against known methods of attack assuming the sandboxed code is not malicious itself. Still, the process is self-contained and does not have access to the external execution context, namely that of the core module.

Dependency Injection: Additionally, built-in `require` or `import` is overridden to control module access and allow requiring dependencies from inside the scripts to be run. Isolation is thus achieved by spawning an independent sandbox for each running WoT System Runtime instance, while importing third-party libraries and built-in modules is still possible from inside. Fig. 4 illustrates the aforementioned sandboxing architecture with the sandbox environment (B) wrapping the WoT System Runtime (A).

C. Execution Control

SRD: The deterministic execution control flow implemented internally and shown in Fig. 3 is directly mapped to and made available by the SRD as Interaction Affordances for consumption. The current active state is exposed by the `status` Property which takes one of the finite list of states: `idle`, `loading`, `running`, `working`, `stopped` or `failed`. Analogously, all transitions between states are represented by Events which fire on state change. The manual state transitions (solid arrows) are exposed via the `start`, `restart`, and `stop` Actions for manual invocation, while automatic state transitions (dashed arrows) remain internal.

Control Mechanism: An additional mechanism for execution control of the sandbox is required since the VM implementation (B in Fig. 4) lacks the ability to stop or kill running processes. Due to this, each sandboxed WoT System Runtime is spawned inside its own worker thread (C in Fig. 4), using the native `worker_threads` library of *Node.js*, which enables using threads to parallelize code execution. Apart from being useful to perform data-intensive operations, workers provide the WoT Runtime with the ability to gain execution control of the WoT System Runtimes by controlling the wrapping worker instance directly, which exposes a `terminate()` method to stop the worker instance as seen in Fig. 4).

[4]https://github.com/patriksimek/vm2

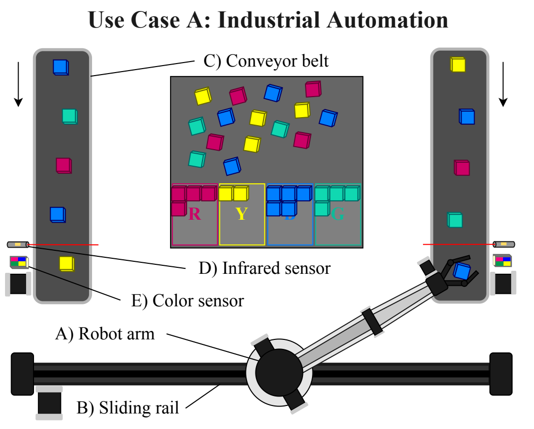
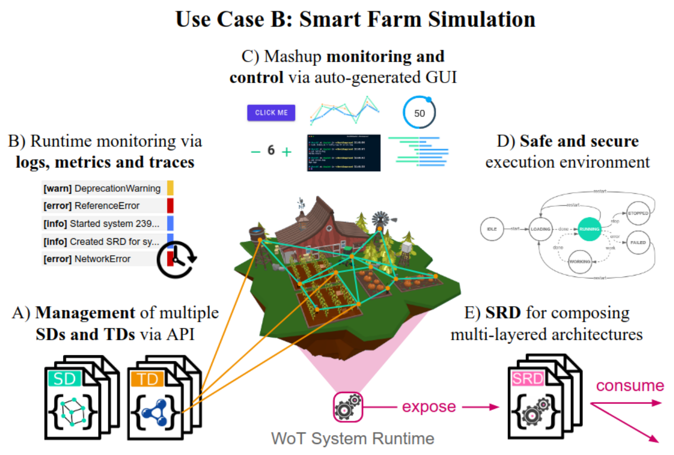


Fig. 5: Top view of industrial automation scenario setup (use case A) with a robot arm (A) mounted on a sliding rail (B) and two conveyor belts (C), each with an infrared sensor (D) and a color sensor (E). The evaluation consists of a WoT System with the task of sorting the blocks by color and placing them in their corresponding drop zone in the middle platform.



Use Case B: Smart Farm Simulation
C) Mashup monitoring and control via auto-generated GUI
CLICK ME
50
− 6 +
B) Runtime monitoring via logs, metrics and traces
D) Safe and secure execution environment
A) Management of multiple SDs and TDs via API
E) SRD for composing multi-layered architectures
SD
TD
SRD
expose
consume
WoT System Runtime

```
[warn] DeprecationWarning
[error] ReferenceError
[info] Started system 239...
[info] Created SRD for sy...
[error] NetworkError
```

Fig. 6: Overview of evaluation results in the example of the smart farm simulation (use case B). Through the WoT Runtime, management of multiple SDs and TDs (A), runtime monitoring via logs, metrics, and traces (B) and mashup monitoring and control via auto-generated GUI (C) is made possible. The WoT System Runtime runs in a safe and secure environment (D) exposed by the SRD for consumption and composition of multi-layered architectures (E).

Apart from gaining an execution control mechanism, the worker adds an additional layer of security to the already sandboxed WoT System Runtime environment. In addition to our implementation, usage of workers could eventually be adopted in browser clients via the web-native WebWorker API, instead of the `worker_threads` module. By doing so, the WoT Runtime could benefit from isomorphic deployments into either the core module (*Node.js*) or the UI module (browser). The optionally isomorphic worker (C) can be seen in Fig. 4 wrapping the already sandboxed environment (B).

V. Evaluation

In this section, we present two case studies to exemplify our contribution and evaluate our approach. These are:

- An industrial automation use case consisting of a robot arm and two conveyor belts, each with an infrared sensor and color sensor, illustrated in Fig. 5.
- A smart farm simulation consisting of different sensors and sprinklers, illustrated in Fig. 6

Metrics: The correct deployment of both WoT Systems in SD format, proper code generation and execution, as well as the ability to manage and monitor the running WoT System Runtime through its SRD were evaluated.

Procedure: The steps taken for the evaluation of each WoT System using the WoT Runtime UI are:

1) Create a new WoT Runtime host with default WoT Servient configuration.
2) Upload an SD to describe the WoT System.
3) Start the system and check for correct Property readings, Action invocations, and Event notifications from the SRD in the UI and directly through an HTTP client.
4) Stop the System, modify its SD, re-deploy and repeat steps 2-3 for the System described by the new SD.

Setup: Evaluation was made on two independent hosts, each running a containerized WoT Runtime process with the default WoT Servient configuration. Table I gives an overview of the setup for each.

Results: Both use cases were evaluated with the aforementioned procedure with positive results. The evaluation proves that the approach and implementation presented by this paper works in practice. With reference to the requirements outlined in Section III.A, we have succeeded in enabling remote deployment, management, and monitoring of WoT System Runtime instances from multiple arbitrary clients (R1), exposing them via the SRD to allow for consumption within the WoT context (R2). The SRD conveniently provides monitoring and verification utilities for both runtime and Mashup-specific information (R3). Moreover, the modular design of the WoT Runtime implementation allows composing loosely-coupled software systems (R4), exemplified by the also implemented WoT Runtime UI (R5), which serves as a reference visual client. A proof of concept is realized in industrial automation and smart agriculture case studies, asserting that an improved version of the proposed implementation could be used in real-life deployments.

The two distinct setups, shown in Table I, indicate that the proposed methodology works for devices with different resource constraints, like the *Raspberry Pi 4* used for the smart agriculture case study and a conventional laptop used for the industrial automation case.

Criteria	**Case Study A**	**Case Study B**
Host OS	*Ubuntu 20.10*	*Raspberry Pi OS*
Protocol bindings	HTTP, CoAP	HTTP
Nb. of Things	5	9
Lines of TS code[a]	1393	252
Lines of JS code[b]	1466	288
Deployment format	SD	Code

[a]TypeScript source code. [b]Transpiled JavaScript code.

Table I: Evaluation setup for the industrial automation scenario (A, illustrated by Fig. 5) and the smart farm simulation (B) (shown in Fig. 6 use cases comparing them based on relevant criteria, listed in the first column.

VI. Related Work

Despite similar methodologies existing for the deployment, management and, monitoring of software systems, only few target the W3C WoT. Our approach remains unique in that it brings together all three features while focusing on the WoT exclusively. We compare related work from the literature in three categories listed below.

Deployment: Several solutions come to mind for the remote deployment of software and firmware in the IoT. For practical purposes, we list the distinct approaches rather than specific solutions. On the cloud infrastructure side, Serverless and Containers are noteworthy. Serverless allows for the remote deployment and execution of functions in an as-a-Service manner, but remains best-suited for event-driven workflows. Containers on the other hand provide an encapsulated runtime for executing arbitrary code, and are thus similar to our proposed WoT Runtime. Nevertheless, container solutions (e.g. Docker[5]) do not provide a mechanism for deploying SDs directly like the WoT Runtime does, and are thus not streamlined for WoT applications. On the IoT side, multiple solutions for over-the-air (OTA) firmware deployment directly to devices exist [8]. These, however, are usually suited for device-specific logic and security patches, rather than WoT Mashups to program interactions between them like we propose with the WoT Runtime.

Specific to the W3C WoT, CLI tools like *WoT Application Manager (WAM)*[6] and the `thingweb.node-wot` library CLI itself exist, which provide a lower-level command-line interface for executing arbitrary WoT scripts at the host. Similarly, the authors at [3], propose different functions to enable the use of SDs for code generation and deployment purposes, but lack any way to systematically monitor such said deployments. On the other hand, the WoT Store proposed at [9] proposes mashup deployment, however, it does not enable multi-layer architectures nor a way of monitoring such deployed mashups.

Management: While many alternative platforms for management of containerized workloads and services exist (e.g. Kubernetes), these are not specific to WoT nor do they come with support for direct deployment of WoT Systems from code or SDs. The WoT Store [9] provides some basic management functionality but is not generalizing this to allow management of any mashup from the same abstraction level. The WoT Runtime provides management features in a way to allow management of any WoT mashup, while also supporting deployment and monitoring needs.

Monitoring: No other monitoring solution was found with direct support for monitoring of W3C WoT applications. Generalizing, traditional software monitoring tools like Prometheus[7], OpenTelemetry[8] could be used to monitor WoT scripts based on logs, metrics and traces. However, these solutions require programmatically handling the monitoring functionality inside the WoT scripts, making them impractical for WoT Mashup development. Moreover, while great for observability at the network layer (e.g. incoming/outgoing network requests), they have no support for monitoring neither runtime nor Mashup-specific information like our proposal. Visual monitoring solutions like Grafana could be used to monitor WoT applications in flexible ways. However, this would require a high level of configuration from the user, as these tools are generally data source agnostic, but not WoT specific nor do they have support for TDs. The WoT Runtime on the other hand leverages TDs directly for zero-configuration, automatic GUI generation of both runtime and Mashup-specific information, making it practical for streamlined WoT development.

[5]https://www.docker.com/

[6]https://github.com/UniBO-PRISMLab/wam

[7]https://prometheus.io/

[8]https://opentelemetry.io/

VII. Conclusion

This paper introduces an additional building block to the Web of Things for the remote deployment, management and, monitoring of WoT Systems. With it, WoT System development is streamlined and maintenance reduced, accelerating the W3C WoT innovation cycle. We have applied our approach and evaluated our publicly available implementation in two case studies: an industrial automation scenario and a smart agriculture scenario, demonstrating how our proposed solution works in practice. In these two use cases, we show how our WoT Runtime enables scalable, reliable, and safe WoT Systems with minimal development effort, bringing the W3C WoT closer to users, reinforcing its adoption, and motivating the advancement into distributed, multi-layered WoT architectures.

References

[1] Web of Things (WoT) Architecture 1.1. M. Lagally; R. Matsukura; T. Kawaguchi; K. Toumura; K. Kajimoto. W3C. 9 April 2020. URL: https://www.w3.org/TR/2020/REC-wot-architecture-2020F0409/

[2] Web of Things (WoT) Thing Description. S. Käbisch; T. Kamiya; M. McCool; V. Charpenay; M. Kovatsch. W3C. 9 April 2020. W3C Recommendation. URL: https://www.w3.org/TR/2020/REC-wot-thing-description-20200409/

[3] A. Kast, E. Korkan, S. Käbisch and S. Steinhorst, "Web of Things System Description for Representation of Mashups," 2020 International Conference on Omni-layer Intelligent Systems (COINS), Barcelona, Spain, 2020, doi: 10.1109/COINS49042.2020.9191677.

[4] D. Guinard and V. Trifa, "Towards the Web of Things: Web Mashups for Embedded Devices," in Workshop MEM 2009, in proc. of WWW, Madrid, Spain, 2009.

[5] Web of Things (WoT) Scripting API. Z. Kis; D. Peintner; C. Aguzzi; J. Hund; K. Nimura. W3C. 24 November 2020. W3C Working Draft. URL: https://www.w3.org/TR/2020/NOTE-wot-scripting-api-20201124/

[6] Sridharan, C. (n.d.). Chapter 4. The Three Pillars of Observability. In Distributed Systems Observability, 2018, O'Reilly Media.

[7] D. Peintner, M. Kovatsch, C. Glomb, J. Hund, S. Kaebisch, V. Charpenay, "Eclipse Thingweb Project", 2018, [Online; accessed April 21, 2019]. Available: https://projects.eclipse.org/projects/iot.thingweb

[8] J. Bauwens, P. Ruckebusch, S. Giannoulis, I. Moerman and E. D. Poorter, "Over-the-Air Software Updates in the Internet of Things: An Overview of Key Principles," in IEEE Communications Magazine, February 2020, doi: 10.1109/MCOM.001.1900125.

[9] L. Sciullo, C. Aguzzi, M. Di Felice and T. S. Cinotti, "WoT Store: Enabling Things and Applications Discovery for the W3C Web of Things," 2019 16th IEEE CCNC, 2019, doi: 10.1109/CCNC.2019.8651786.