Exploring software architecture trends in the Internet of Things

REED KLAESER, Depaul University, United States

This paper explores emerging software architecture trends in the Internet of Things (IoT), focusing on event-driven architectures, edge computing, and containerization. It analyzes the benefits and drawbacks of each approach, highlighting their impact on resource efficiency, real-time processing, and maintainability while identifying key challenges in coupling, complexity, and security.

Additional Key Words and Phrases: IoT, Containers, Event-Driven Architecture, Edge Computing

ACM Reference Format:
Reed Klaeser. 2025. Exploring software architecture trends in the Internet of Things. 1, 1 (March 2025), 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Internet of Things systems combine aspects of embedded and distributed systems to form locally-aware, real-time, internet-connected networks. These attributes have driven the rapid growth of vast networks of interconnected devices in both industrial and consumer settings that we now call IoT. [Statista 2025] estimates that by this year there will be 20 billion connected devices (seen in Figure 1). This rapid expansion raises questions. Are all these new devices a huge security liability? Questions of opportunity arise as well. Can real-time systems reduce latency by intelligently distributing work within the network? And old assumptions are questioned. Is a request-response architecture sufficient or is an event-driven one a better fit?

Popular architecture trends offer answers to these questions. An event-driven architecture proposes a more timely allocation of resources than a request-response architecture. Edge computing supposes it can decrease latency by pushing work to the edge of networks. And containerization makes its claim as an easily orchestrated deployment strategy with other advantages to boot.

This paper explores the benefits and drawbacks of these three architecture trends, analyzing how they influence resource efficiency, real-time data processing, and security in IoT systems. Specifically, we examine:

- **Event-driven computing**: How does an event-driven approach enhance efficiency while managing trade-offs in data persistence?
- **Edge computing**: What specific strategies does edge computing offer to divide work in a network?
- **Containerization**: How does containerization's implementation set it apart from competing deployment patterns?

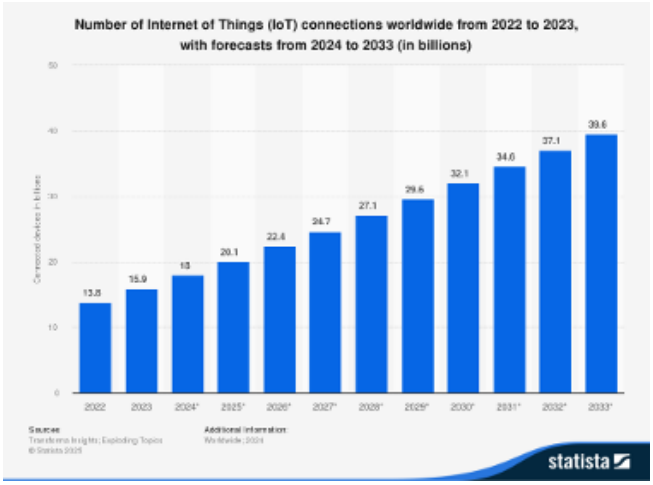

Fig. 1. Forecasted growth in number of IoT connections (connected devices), via Statista.

By evaluating these architecture trends, we aim to highlight key challenges and opportunities in IoT architecture and identify potential directions for future research and development. The remainder of this paper can be outlined as follows. Section 2 explains the varying demands of IoT use cases. Section 3 explains the components of an IoT system. Sections 4 through 6 investigate event driven, edge computing, and containerized architectures and Section 7 concludes with a future direction.

2 IoT Use Cases

IoT systems have three primary use cases that are worth understanding because of the differing demands they create on system uptime and data completeness. They are alerting, real time decision making, and data collection.

Thanks to IoT, jet engines now have digital twins that help alert when they need maintenance. Engine manufactures, who are paid per mile for their engines need to predict when maintenance is necessary and when it's better to keep engines in operation. This turns out to be a function of many characteristics including real time data like the temperature and pressure experienced by certain components. Digital twins keep track of this data and help tune models that decide when certain engines get maintenance and even run simulations that help them allocate resources for the future. Consumer IoT alerting applications exist too, as in a smart watch that warns of an impending cardiac event. A characteristic of these systems is that they must be ready to alert at any times but do not need to be in active communication at all times.

Thanks to IoT, many manual machines are now automaton, that is capable of real-time self-operation. Electric grids face heavy strains in the hot summer months, especially at peak hours when people return from work. Smart thermostats can be programmed to cool

Author's Contact Information: Reed Klaeser, Depaul University, Chicago, United States.

ACM XXXX-XXXX/2025/3-ART
https://doi.org/XXXXXXX.XXXXXXX

homes before peak hours. Additionally, smart grids collect real time demand information and adjust supply accordingly. Consumer automaton exist too thanks to IoT. An insulin pump, for instance, might decide when and how much insulin to deliver to its users via algorithms that are tuned with data from millions of users. A characteristic of these systems is that their decision making can be improved by cloud processing of large data but the decisions themselves must come rapidly and often that means locally.

And finally IoT supports data collection for as needed decision making processes. Inventory numbers might be collected from various distributed locations allowing a centralized decision maker to make new orders as needed. A characteristic of these systems is that a massive amount of data will be generated and should be parsed so that only data relevant to decision making is kept.

3 An Overview of the IoT System

This section explores the devices that compose an IoT system and discusses the constraints these devices face.

Servers in a data center are powerful, but they are stuck in place. They need nodes to collect data on the real world. Still, servers are often too far away to act quickly on this data. What is more, servers would often have trouble communicating with nodes without translators that can understand both local protocols like MQTT and internet protocols like HTTPS. We have thus described the essential devices of an IoT system. Remote servers, often in the cloud. Nodes that collect data. And edge servers, also commonly called gateway devices because of their role translating protocols. The cloud server is constrained because it is far away. The nodes have minimal processing because of their low-spec chips and frequent reliance on battery power. Some nodes are further constrained by availability requirements because of their use in real time systems. Edge servers have neither the extreme constraints of nodes nor the extreme power of cloud servers and are geographically distributed and co-located with nodes, a property that serves as both a constraint and an advantage.

In summary, an IoT system is composed of three layers of devices: remote servers, edge servers, and nodes and these layers are subject to different constraints that drive their need to for the solutions offered by our three architecture trends. Each architecture trend we discuss has applications for multiple layers of the system. To illustrate, events can be sent between nodes and edges as well as edges and remote servers. Similarly, edge computing can involve pushing work from remote servers to edges or from edges to nodes. And finally, containerization can be applied to both nodes and edges.

4 Event Driven

As described in IoT uses cases, alerting demands that a system be always ready but not always on. An event-driven architecture can be contrasted with continuous methods of sending and receiving data, respectively data streams and continuous polling. In contrast to streams, events send data intermittently. Less data is therefore sent over the network. Sending less data may involve waiting for significant changes, or processing data streams on nodes and sending back processed data in events. We'll discuss the strategies for processing data on nodes in the edge computing section. Even if

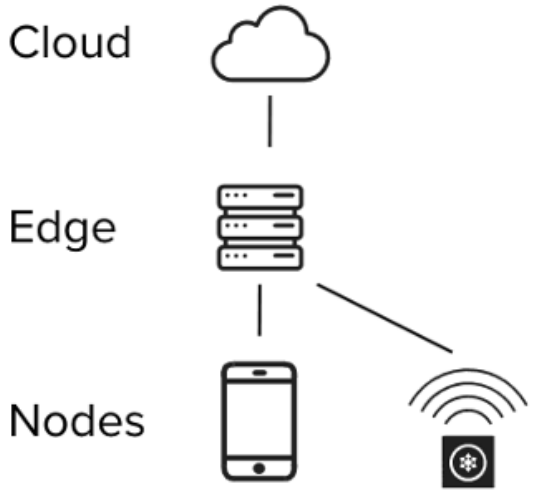

Fig. 2. Nodes, Edges, and Cloud: the devices of IoT architectures

less data is sent, there is still enough data that a strategy needs to exist for sorting data by priority. [Al-Osta et al. 2019] explains how a semantic model similar to that used to classify terms on the web can be used to classify the priority of events.

In contrast to continuous polling, processes waiting for data are notified so they don't have to continuously check. The CPU is then released to do other work and an interrupt notifies it to go back to the waiting process. Real time operating systems, like FreeRTOS implement specialized task scheduling algorithms that help prioritize the most important tasks. Unnecessary CPU cycles are avoided and energy is saved. In contrast, a traditional multitasking operating system gives the highest priority to jobs with the lowest demand on the computer. In an IoT environment, the network processing and energy savings owed to an event-driven system are immensely valuable.

Event-driven architectures don't stop there. Now that sending and receiving processes are no longer in constant communication, in other words, they are decoupled, the sending receiving relationship no longer needs to be exclusive. Instead, event producers can send messages to multiple event subscribers. This concept is closely linked to the pub/sub pattern and other system decoupling patterns. Consider a workflow for when a package is scanned on its way out of a warehouse. The sensor could send an event to an edge server that hosts containers for several subscribers. A UPS service will let UPS know the package is on the way to their facility, a customer notification service will let the customer know that the package has left the warehouse, and an accounting process will keep track of order status internally. When events are sent to multiple subscribers, there are two choices for what information is shared. The event can contain all the relevant information so that subscribers can act on it right away or the event can contain the primary key which subscribers can use to look up the rest of the data. Overstuffing events with data only relevant to some subscribers is essentially coupling

the subscribers together. The primary key approach ensures that events are decoupled but will be slower because several extra steps are required to write and read data before the subscriber can act [Microsoft 2025].

5 Edge Computing

As we've discussed some IoT uses cases demand real time decision making. Edge computing is the idea that processing should be moved closer to the periphery of a network. This lets systems send less data over the network and decrease latency. Edge computing includes both moving remote server processing to the edge and edge processing to nodes. When different layers of the network have different owners, edge computing also increases privacy and security. Federated learning is a particular flavor of edge computing with privacy and security benefits. It works by training models from node devices without nodes sharing raw data. A classic example of federated learning, training a hospital's heart monitors, reinforces the idea that edge computing can be achieved at both the layer of nodes and edges. In this system heart monitor nodes record data, edge servers collect it, and remote servers apply learning models to it. You can imagine that a model trained on multiple hospital system's data might do a better job of predicting an imminent heart event but HIPAA requirements would obviously prevent such a system. However, if federated learning were applied, heart monitors could do an initial layer of processing, sending only time bounded data around detected events and using anonymous IDs rather than identifiable information. Edge devices could then do an initial level of processing and send intermediate data back to the cloud to be aggregated and processed further. [H. Li and Dong 2018] explains how "it is very hard to understand the original information with the features extracted by a convolutional neural network (CNN) filter in the intermediate CNN layer". [H. Li and Dong 2018] Also explain how this works by showing that the lower levels of a CNN can be performed by edge devices and the higher level on cloud devices.

[H. Li and Dong 2018] also describes algorithms for distributing processing among nodes in a network. This could take the form of moving processing from nodes with limited battery supply to nodes with significant power resources or using map reduce to divide processing among a network of nodes [et al. 2018]. [B. Chen and Zhang 2018] describes real-time data fusion that allows industrial IoT devices to communicate with each other to ensure that anomalous readings are not acted on.

These optimizations are not without their tradeoffs, division of processing increases the complexity of applications and creates distributed debugging problems that make root causes difficult to nail down. Patterns and tools should be developed to manage this complexity.

6 Containers

Containerization is a deployment strategy where applications run partially isolated on virtual machine. Containers are isolated from each other using Namespaces, Chroot, and Cgroups but still share the host OS kernel of the virtual machine. Technically, there exist both single application containers, called application containers, and multi-application containers called system containers but our discussion will focus solely on application containers which are both faster, more isolated, and more common [Gamess and Parajuli 2024]. In IoT, containerized environemnts can be contrasted with firmware, bare metal, virtual machine, and unikernel environments. Containers are created by layering an application's dependencies together to form an image that can be deployed. The first layer is frequently a lightweight Linux distribution, which is deployed in the container for its libraries, rather than its services. Remember, the host OS is providing the OS services. This can be confusing since a commonly stated advantage of containers is that they are lighter weight than virtual machines. This is still true, since the minimal need for operating systems in containers has allowed operating systems for containers to be shrunk way down, for instance Linux Alpine is around 5MB and BusyBox is 2MB compared to Ubuntu at 188 MB [Christner 2025]. For most IoT systems, however, the more important lightweight characteristic of containers is that not running a second operating system reduces CPU and RAM. Relying on the OS as a library allows it to get very small. After the base layer, application dependencies, runtimes, and the app itself form their own layers. Layers help reduce the footprint of containers because shared layers can be reused between containers running on the same machine.

Containers can be used on both IoT nodes and edges. They are commonly contrasted to virtual machine deployments but since this is IoT we'll first discuss firmware on nodes. Firmware provides low-level control over hardware, such as storage, input devices, and peripherals. When a computer with an operating system is turned on, a piece of firmware called the BIOS starts up another piece called the bootloader that in turn starts the operating system. When a firmware only IoT device runs, both the startup logic and the application logic are stored within the firmware. Fewer layers means fewer resources but what if a firmware update goes wrong. Now you've broken the system that starts the machine, how do you start up the machine to fix the problem? If a firmware update were to require manual intervention, this would be especially difficult in an IoT environment where devices can be distributed across warehouses or in the control units of wind turbines. Today, there are tools to safely rollback firmware updates like A/B deployments but firmware updates are still considered riskier than software updates. Additionally, firmware updates require rebooting the machine to switch over to the new firmware whereas software updates like those to containers can maintain continuous uptime using A/B deployments.

Bare metal machines are computers with an operating system running directly on hardware. They involve more overhead than firmware because of the BIOS, bootloader, and operating system described above but they are safer to update since their BIOS does not need to be modified. Their direct interfacing with machine hardware means that they are more performant than either virtual machines or containers but that, like firmware devices, they cannot be cloned. Cloning is a distinct advantage because it means an operating system can be configured once and copied to all IoT devices in a fleet. This is a distinct disadvantage in IOT, where a fleet can comprise thousands of devices.

Virtual machines run on a hypervisor that takes over hardware-centric OS tasks like managing memory, time sharing a CPU, network, and drivers. By abstracting hardware away, multiple machines can now be run on a single bare metal machine and machines can be cloned and deployed elsewhere. As discussed above, this is of particular importance in an IoT setting. Each virtual machine is a full operating system with their own kernel so multiple machines on a single hypervisor are largely isolated from each other. Hypervisors do occasionally have vulnerabilities that expose one virtual machine to another but they tend to be less common than operating system vulnerabilities [Gonçalves et al. 2025].

As described above, containers are partially isolated. Namespaces and Cgroups create some isolation but as [Casalicchio and Iannucci 2020] points out, the reliance on shared OS services, in particular the same network bridge is a hole in container isolation. Moreover [Casalicchio and Iannucci 2020] says, "container isolation can be lowered at launch time playing with specific settings". Despite this incomplete security, containers popularity in cloud environments has overflowed into increasing use for IoT edges and nodes. With the growth comes an influx of new tooling, for instance, the energy usage of containers can now be managed with DockerCap to ensure it is within the strict requirements of IoT devices [Asnaghi A. 2016]. And [Dolui and Kiraly 2018] show how the multi-functional requirement of Iot Edge/Gateway devices make them a perfect candidate for multi container deployments to manage "discovery, data management and cloud integration". Further, [A. S. Gaur and Lung 2018] show how DockerCompose supported there project to create a multi-container IoT gateway capable of mobile handoffs, which are necessary to pass the application state of mobile nodes between gateways. Also working in the domain of service handoff, [Lele Ma and Li 2017] showed that containerization can reduce service handoff time by 80% with network bandwidth of 5Mbps. Finally, [Watada 2019] shows in Figure 3 that a containerized solution (Docker) gives the closest to native performance compared to virtual machines (kvm, Xen), and unikernels (OSv, Rumprun). [Watada 2019] goes on to say that with maturity, unikernels are expected to improve.

	Execution time (in *secs.*) for following number of threads							
	1		2		4		8	
Native	11.21		11.30		11.25		11.27	
	ES1	*ES2*	*ES1*	*ES2*	*ES1*	*ES2*	*ES1*	*ES2*
Docker	19.35	31.32	19.33	31.37	19.34	31.67	19.36	31.39
OSv	37.56	42.46	37.81	42.71	37.72	42.52	37.63	42.91
Rumprun	35.91	40.21	35.82	40.35	35.78	40.76	35.77	40.12
LXC	25.45	32.57	25.46	32.82	25.44	32.37	25.40	32.15
rkt	30.34	32.39	25.36	32.31	25.34	32.36	25.31	32.33
kvm	31.35	31.21	31.56	31.65	31.34	31.13	31.38	31.32
Xen	31.32	31.12	31.43	31.27	31.56	31.41	31.48	31.24

Fig. 3. CPU performance comparison between native, containers, virtual machines, and unikernels, via Watada2019.

Unikernels are minimalist virtual machines that, like virtual machines, run directly on hypervisors. They offer the same level of isolation as virtual machines but achieve the lightweight attributes of containers. As a consequence of stripping down the OS, unikernels tend to moot extremely fast [Watada 2019]. Unikernels are a natural progression from containers. If applications are now run in single-tenency environments, then why pay the overhead cost of multi-user memory spaces and the time sharing. Pairing down these features of operating systems leads to much smaller memory footprints, a few MB, instead of hundreds of MB. The LynxOS unikernel has been used in high-performance and low error tolerance environments like fighter jet cockpit software. However, unikernels are a developing technology and [Talbot et al. 2020] shows that some distributions have not yet implemented basic buffer overflow protections or address space memory randomization. Finally, unikernels lack the wealth of tools to build, configure, and deploy that containers and virtual machines have. Nonetheless, recent work to show how the Linux kernel can be made unikernel friendly shows that unikernels may see continued growth [Raza et al. 2019].

Given the deployability and relative performance benefits of containerized applications, despite their security concerns, they should be seen as the de-facto environment for edge gateways and node devices in IoT. When deployability is not a concern, native/bare-metal environments offer the best performance and unikernels should be monitored for future application.

7 Conclusion and Future Direction

This paper covered three architectural trends in IoT systems. Event-driven architectures were shown to be effective means of reducing network and processing. Edge processing was shown to have applications in allocating processing between, as well as within, IoT layers. And containers were shown to be the go-to choice for IoT application deployments. Future research should focus on expanding the applications of event-driven architectures from IoT alerting, and training to real time decision making. Additionally, it should investigate patterns for managing the complexity resulting from edge computing. Finally, security improvements to containers should be investigated alongside performance optimizations for unikernels.

References

J. Budakoti A. S. Gaur and C. H. Lung. 2018. Design and Performance Evaluation of Containerized Microservices on Edge Gateway in Mobile IoT. In *2018 IEEE International Conference on Internet of Things (iThings).* doi:10.1109/Cybermatics_2018.2018.00055

M. Al-Osta, A. Bali, and A. Gherbi. 2019. Event driven and semantic based approach for data processing on IoT gateway devices. *J Ambient Intell Human Comput* 10 (2019), 4663–4678. doi:10.1007/s12652-018-0843-y

Santambrogio M. D. Asnaghi A., Ferroni M. 2016. DockerCap: A Software-Level Power Capping Orchestrator for Docker Containers. In *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC).*

A. Celesti D. Li H. Abbas B. Chen, J. Wan and Q. Zhang. 2018. Edge Computing in IoT-Based Manufacturing. *IEEE Communications Magazine* 56, 9 (2018), 103–109. doi:10.1109/MCOM.2018.1701231

Emiliano Casalicchio and Stefano Iannucci. 2020. The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and Computation: Practice and Experience* 32 (2020). doi:10.1002/cpe.5668

Brian Christner. 2025. Docker Image Base OS Size Comparison. https://brianchristner.io/docker-image-base-os-size-comparison/ Accessed: March 16, 2025.

K. Dolui and C. Kiraly. 2018. Towards Multi-Container Deployment on IoT Gateways. In *2018 IEEE Global Communications Conference (GLOBECOM).* doi:10.1109/GLOCOM.2018.8647688

W. Yu et al. 2018. A Survey on the Edge Computing for the Internet of Things. *IEEE Access* 6 (2018), 6900–6919. doi:10.1109/ACCESS.2017.2778504

Eric Gamess and Mausam Parajuli. 2024. Containers Unleashed: A Raspberry Pi Showdown Between Docker, Podman, and LXC/LXD. *SoutheastCon 2024* (2024). doi:10.1109/SoutheastCon52093.2024.10500266

Charles F. Gonçalves, Xavier Mendes, and Marco Vieira. 2025. *Hypervisors Vulnerabilities Analysis: Causes, Effects and Consequences.* https://www.cisuc.uc.pt/download-file/16657/XBPb4BAII7BIFHqE47Tz

K. Ota H. Li and M. Dong. 2018. Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing. *IEEE Network* 32, 1 (2018), 96–101. doi:10.1109/MNET.2018.1700202

Shanhe Yi Lele Ma and Qun Li. 2017. Efficient service handoff across edge servers via docker container migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing (SEC '17)*. doi:10.1145/3132211.3134460

Microsoft. 2025. Event-Driven Architecture Style. https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven Accessed: 2025-03-17.

Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. 2019. Unikernels: The Next Stage of Linux's Dominance. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 7–13. doi:10.1145/3317550.3321445

Statista. 2025. *Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions)*. https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

Joshua Talbot, Przemek Pikula, Craig Sweetmore, Samuel Rowe, Hanan Hindy, Christos Tachtatzis, Robert Atkinson, and Xavier Bellekens. 2020. A Security Perspective on Unikernels. In *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*. 1–7. doi:10.1109/CyberSecurity49315.2020.9138883

Junzo et al. Watada. 2019. Emerging Trends, Techniques and Open Issues of Containerization: A Review. *IEEE Access* (2019). doi:10.1109/ACCESS.2019.2945930

Received 20 March 2025