NexusEdge: Leveraging IoT Gateways for a Decentralized Edge Computing Platform

Nabeel Nasir
University of Virginia
nabeeln@virginia.edu

Victor Ariel Leal Sobral
University of Virginia
sobral@virginia.edu

Li-Pang Huang
University of Virginia
lh5jv@virginia.edu

Bradford Campbell
University of Virginia
bradjc@virginia.edu

***Abstract*—Edge computing enables scalability and privacy improvements for Internet of Things (IoT) systems, by shifting applications from the cloud to edge servers closer to IoT devices. Conceptually, IoT devices communicate directly with the edge, but in real-world IoT deployments often IoT gateways are needed to bridge devices and edge servers. Design decisions at this gateway layer directly contribute to the responsiveness of edge applications and scalability of the platform, yet these gateways are often overlooked and under-explored. IoT gateways have a compelling mix of features, including reasonable compute capabilities, low cost, direct contact with devices, and spatial distribution in deployments. We hypothesize that a new management layer that organizes already existing gateways can replace expensive edge servers while enabling the privacy, reliability, and performance benefits of executing IoT applications on the edge.**

We utilize a decentralized architecture that creates a nexus among disjoint gateways using out-of-band discovery, low-overhead abstraction layers, and runtime application scheduling. This platform supports heterogeneous devices, minimizes configuration overhead, executes applications, and provides resiliency to failure. We develop a prototype of the architecture, NexusEdge, and deploy it across several gateways and hundreds of low-power and energy-harvesting devices. When compared to Amazon's AWS IoT Greengrass, NexusEdge shows a 10x improvement in application latency, and a 2.5x reduction in network traffic, indicating better scalability and responsiveness. We demonstrate how NexusEdge supports applications without cloud support, and envision future extensions of this platform.

***Index Terms*—Edge Computing, IoT, Gateways, Middleware**

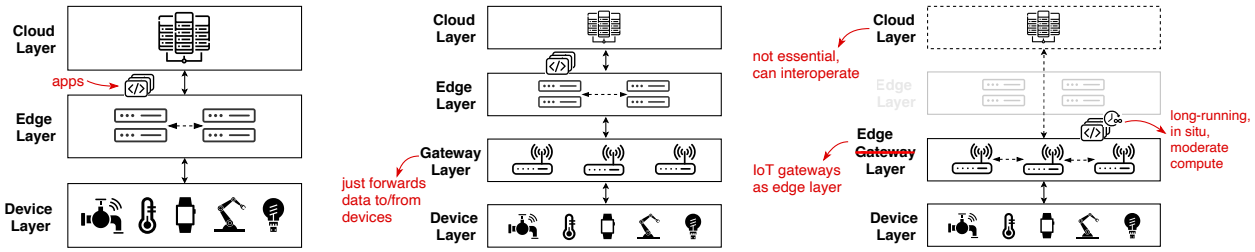
I. Introduction

The Internet of Things (IoT) is growing at a rapid pace; projections indicate that there will be more than 41 billion IoT devices by 2025, generating a colossal 79 billion zettabytes of data [1]. Edge computing is a potential solution to handle the scale of this growth, which prescribes executing applications closer to devices, consequently reducing latency, minimizing bandwidth, and improving privacy by operating on premises. Edge computing platforms make it possible for developers to deploy applications closer to devices while retaining cloud support. These platforms typically follow a three-tier architecture, as shown in Figure 1a, with sensors and actuators at the bottom layer, edge nodes in the middle layer (usually server machines), and cloud data centers at the top.

With the proliferation of low-power and battery-less devices, IoT devices often employ short-range radios like Bluetooth Low Energy (BLE), rather than 5G or WiFi, to conserve power. Since edge server machines are not equipped with the requisite radios, and even so deploying a large number of edge servers to provide dense, short-range wireless coverage is cost-prohibitive, IoT gateways are used instead to bridge this gap between devices and edge servers, as depicted in Figure 1b. Although the gateways are necessary, they are usually considered a part of the implementation rather than the design. However, how gateways are used can have a significant impact on the scalability and responsiveness of edge applications built on IoT sensors and actuators. We take a new approach, and study the gateway layer from a design perspective, specifically exploring the feasibility of extending edge computing to better exploit these gateways, and ultimately increasing the reach of the edge-centric computing.

Increasing the responsibility of edge gateways, e.g. running applications directly on the gateways rather than on edge servers, is counterintuitive. Edge servers offer more resources for applications including storage and memory, heavy compute (CPU and GPU), and are less prone to failures. However, we make five observations that suggest edge computing can benefit from leveraging these gateways, particularly when applications are using IoT devices. First, since gateways have more deployment flexibility and include wireless radios to communicate with devices, applications running on gateways can operate one hop from sensors and actuators, and leverage protocol-specific information for optimizations. Second, streaming all data to centralized edge servers increases network traffic and application latency. If applications can be executed on gateways instead, this overhead can be reduced. Third, advances in single board computers like the fourth generation Raspberry Pi [2] have made IoT gateways increasingly performant. Gateways are capable enough to support containerized environments [3], and are increasingly being used for various kinds of edge applications [4], [5], [6], [7], [8]. Fourth, edge servers can be costly (exceeding $1,000), and leveraging gateways can make edge computing feasible for cost-conscious deployments. Finally, spatially large IoT deployments already have multiple gateways for network coverage, and leveraging them increases their value without incurring additional hardware and installation costs.

Using gateways for edge computing may not be suitable for all classes of applications, but we argue there is a nontrivial set of applications well suited for gateways. Applications executing in situ on an edge node, requiring only low/moderate amounts of compute power (CPU, GPU, memory) and storage,



(a) Edge Computing Architectures

(b) Actual implementations use gateways

(c) Shift applications to gateways

Figure 1: a) Edge computing generally follows a layered architecture in which apps execute on edge servers near devices. b) Actual deployments require IoT gateways to communicate with devices. c) We leverage gateways to execute apps without edge servers or the cloud. Apps execute fully on gateways, run indefinitely, and have moderate compute and storage requirements.

and run indefinitely once deployed, are good candidates to execute on gateways. Applications that sense and act in large-scale IoT deployments, simple if-this-then-that (IFTTT) applications, machine learning at the edge, and inherently distributed applications all meet these criteria. A prototypical IoT application such as triggering alerts for anomalies in plug-level appliance power data in a factory [9] is an example where the application can execute on the edge, requires moderate amount of compute power to handle streaming data, and executes indefinitely. We do not focus on compute-intensive tasks such as live video analytics and augmented reality.

We hypothesize that an edge computing platform built using a network of IoT gateways can support and scale up with responsive edge applications and cutting-edge IoT devices, and eases application development, without requiring cloud support (Figure 1c). To convert a disjoint network of gateways into a cohesive platform, we build a thin management layer which can handle application load balancing, optimize latency and network communication overhead, provide automatic discovery and scalability, ensure resilience to failures, and operate autonomously without any Internet connectivity. Then, we design an open-source edge computing platform, NexusEdge, which aims to manage heterogeneity, minimize configuration, aid in network management, and support applications.

We describe the key design decisions for our IoT middleware learned from iterating our platform and interfacing with a variety of devices and applications. Our key design choices include: i) Remaining agnostic to specific wireless protocols and data formats to not artificially restrict use cases and future development. We show through our layered approach how data models can be applied for specific deployments. ii) Relying on out-of-band wireless discovery for local gateway coordination. This eliminates the *requirement* for cloud interaction and promotes scalability. iii) Using an intentionally thin API between the platform and applications to allow seamless access control of data, load balancing, and scaling. Again, we choose a design focusing on the core edge computing platform, leaving more sophisticated application support to pluggable software layers.

We showcase the suitability of NexusEdge by deploying it across five gateways spanning a floor of a building and supporting 181 low-power and energy-harvesting devices. We

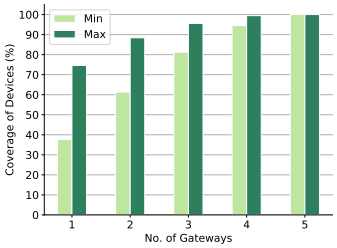


Figure 2: Device Coverage

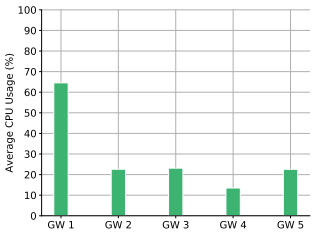


Figure 3: Gateway Load

evaluate NexusEdge by comparing it with a centralized architecture and show improved application performance and better gateway utilization at reasonable decentralization overheads. We demonstrate various applications using the platform, and present a case study of a federated machine learning approach that requires no cloud support to show how NexusEdge eases the burden of developing complex IoT applications. We compare the performance of our platform with Amazon's AWS IoT Greengrass [10], and show improvements in sensor data latency and network traffic, indicating better scalability, and highlighting the benefit of a distributed architecture. We also demonstrate that the middleware stays resilient to failures, ensuring better reliability operating with gateways. These experiments collectively validate the feasibility of using IoT gateways to build an edge computing platform.

Our work provides two key contributions. First, to the best of our knowledge, our work is the first to demonstrate the feasibility of using IoT gateways to build a cohesive platform that can execute edge computing applications. Second, we present the design and implementation of an open-source, cloud-independent, edge computing platform, which can handle device heterogeneity and access control of data, and interoperates with other edge computing platforms. These contributions help expand the vision of edge-centric computing by enabling new use cases and opportunities for spatially coupling data and compute. Specifically for the case of low power IoT devices, the NexusEdge design increases the opportunity for leveraging edge computing ideals in a vast number of IoT deployments that already use gateways to support the devices.

II. Motivation

Our motivation for this work stems from our deployment experience of setting up a testbed of 181 wireless IoT devices in the floor of a building spanning 17,000 sq.ft. As our IoT devices have limited transmission ranges and were spread across a large area, we deployed five gateways to collect data from the devices and send it to a cloud service. We conducted some preliminary experiments with the gateways to study the number of devices they covered and their CPU loads.

Figure 2 shows the minimum and maximum number of devices that can be covered with different combinations of our gateways. We observe that for our deployment, a single gateway at best can cover 74% of devices. To cover more than 95% of the devices, we need at least 3 well-positioned gateways, and 5 are required for full device coverage. This suggests that **multiple gateways are required to cover IoT devices** in moderate-sized deployments like ours.

We then investigated the workload of the gateways in our deployment, which were collecting sensor data on multiple wireless interfaces, formatting data packets, and publishing data to a cloud time-series database. We measured the average CPU usage of our gateways for 60 days and the results are shown in Figure 3. We noticed that most gateways are lightly loaded and spend around 80% of their CPU time idle, with the exception of Gateway 1 which is fetching additional data streams from four different web APIs. This suggests that **IoT gateways have underutilized computational power**.

These insights indicate the availability of multiple underutilized gateways in IoT deployments, which could be leveraged for edge computing.

III. Related Work

Collaborating IoT Gateways: Clemente et al. use a mesh network of gateways to cooperate and provide services to each other [11]. Although similar in their focus on IoT gateways and inter-gateway cooperation, they do not describe the interface between devices and the platform, or applications and devices, which are key components of our design. Ooi et al. present an architecture in which gateways coordinate to provide additional routes from devices to the cloud to improve reliability [12]. This leverages multiple gateways and uses a similar gateway discovery mechanism, but applications are all executed on the cloud, and gateways only cooperate to reliably deliver data to the cloud. In contrast, NexusEdge utilizes gateway cooperation to execute applications and facilitate device interaction, without any cloud support.

Comparison with other Edge Computing Platforms: Cloud platforms provide edge computing solutions such as AWS IoT Greengrass [10] and Azure IoT Edge [13] to execute edge applications. We identify four shortcomings of these platforms. First, they assume a rather simple centralized architecture in which all device data is available at a central edge node and applications execute only on this node. Second, there is no access control of device data, exposing all available data to all applications. Third, they require a cloud connection for configuration and application deployment, restricting use cases with unreliable Internet connectivity. Finally, deploying an application that interacts with even tens of devices require substantial configuration, making it intractable at scale [14]. Our work, in contrast, avoids requiring a centralized node, enforces access control by restricting the data applications have access to, operates without Internet connectivity, and provides convenient abstractions for setting up gateways and devices to reduce development burden.

Handling Device Heterogeneity: IoT devices come with a wide range of wireless radios, network protocols, and data formats, which complicate application development. SemIoTic [15] maps semantic user commands to device actions by abstracting the underlying device heterogeneity with a *DeX API* that provides support for protocols including CoAP, MQTT, and XMPP. However, this excludes resource-constrained devices which cannot support these application protocols. TinyLink 2.0 [14] is a programming language for IoT which automatically generates programs and configuration for the cloud, device, and mobile layers. However, they require devices to be programmable to support specific functions which is not always feasible, and is difficult to scale as the device API varies based on the underlying device. Instead, NexusEdge does not assume devices to be programmable, and supports modules to interact with all types of devices.

Middleware for IoT: There are some prior works on IoT middleware which address challenges such as heterogeneity and resource sharing. ThingsJS [16] can schedule IoT applications on heterogeneous edge hardware. They assume that sensors and actuators can run Javascript code to support device interaction for applications, which excludes low-power and energy-harvesting devices. The middleware also require developers to create and maintain MQTT topics for interacting with devices. Instead, NexusEdge supports software modules on gateways to interact with devices, and reduces developer burden by means of simple APIs to get data streams from devices. The Hive middleware [17] decouples applications, sensors, and processors to maximize resource utilization at the edge. But they do not support actuator devices, is not resilient to gateway failures, and their application scheduling is optimized to reduce energy consumption which may not be relevant for wall-powered edge gateways. Our middleware supports both actuators and sensors, can migrate applications or provide alternative data streams if gateways fail, and schedules applications based on device data locality to reduce network traffic and minimize application latency.

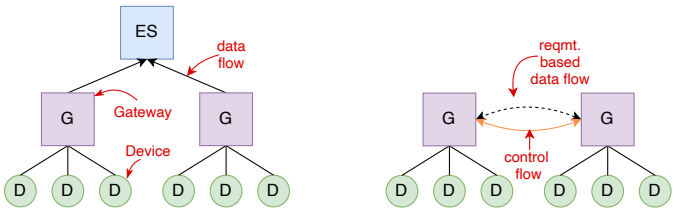
IV. Motivating Applications

To design our edge computing platform and to set the scope of our work, we first categorize IoT applications based on four features, namely execution model, compute requirements, storage requirements, and lifecycle.

We focus on IoT applications that would traditionally execute in the cloud, and explore how they would either run on an edge node, or offload tasks to the edge node. Applications can have low, moderate, or high compute requirements. Gateways (typically single-board computers like the

Application Type	Distinguishing Features				Other Features	
	Execution	Storage	Compute	Lifecycle	Latency	Distributed
Simple IFTTT	in situ	low	low	indefinite	10ms	no
Sense and actuate	in situ	low	moderate	indefinite	100ms	no
Machine learning	in situ	moderate	moderate	indefinite	10ms	no
Inherently distributed	in situ	low	moderate	indefinite	100ms	yes
Compute-intensive	offloaded	high	high	short	10ms	no

Table I: NexusEdge can support applications which operate in situ on gateways, has moderate storage and compute, and is long running (supported applications types are marked green).



(a) Centralized Edge Server Architecture

(b) Decentralized Gateway Architecture

Figure 4: Our work proposes shifting from a centralized server based architecture to a decentralized network of gateways.

Raspberry Pi) cannot provide high compute because of CPU, GPU, and memory constraints. So we focus on low compute complexity apps, and leave applications with high compute requirements such as live video analytics [18], and augmented reality [19] to edge server machines. Applications may require low, moderate, or high data storage requirements. Since current gateways offer only moderate amounts of storage, applications that require long term storage like historical data analysis and visualization [20] would be a better fit for server machines or the cloud. However, current gateways support flash storage up to hundreds of gigabytes [21], making them suitable for low and moderate storage applications. Applications can either execute for a short duration or run indefinitely. Typically IoT applications are long-running, and we focus on those. Applications supported by our platform are categorized into four types based on these characteristics in Table I.

Simple if-this-then-that (IFTTT) applications. These type of applications process data from a few sensors, observe some event, and actuate devices. They can operate in situ, on streaming data with limited to no storage requirements, and with low compute requirements, but require low latency. For example, an app that controls the window blinds based on the time or weather [22], or one that controls the lights when a smart door lock is unlocked [23].

Sense and act in large-scale IoT deployments. These applications use tens or hundreds of sensors and actuators and require multiple gateways to interact with devices. They can operate in situ, have limited to no storage requirements, moderate compute requirements, and usually don't require low latency. Examples include an application which performs condition-dependent agricultural irrigation using multiple sprinklers and soil moisture level sensors [24], [25], and an app that performs building-level energy forecasting with power-meter data and room-level occupancy data in a building [26], [27].

Machine Learning at the Edge. Recent advances in machine learning techniques have enabled fast predictions on constrained hardware without compromising on accuracy [28], [29], [30], [31], [32]. Examples of machine learning applications for IoT include fall detection of elderly patients [33], real-time object detection [34], and greenhouse temperature forecasting [35]. Such applications operate in situ, require low to moderate amounts of storage to store sensor data, have moderate compute requirements, and operate at low latencies.

Inherently distributed applications. Some IoT applications *must* be spatially distributed and cannot run in a centralized manner. For example, SeamBlue [36] provides cellular-like handover functionality to BLE devices, and must run on spatially distributed gateways to transfer BLE connection state as the BLE device moves throughout a space. Similarly, IoT services for wireless devices including time updates [37], firmware updates [38], and fault detection heartbeat messages [39] require gateways near devices that can provide those services. These applications operate in situ but span multiple gateways, with limited to no storage requirements, moderate compute, and usually do not have low latency requirements.

V. System Design

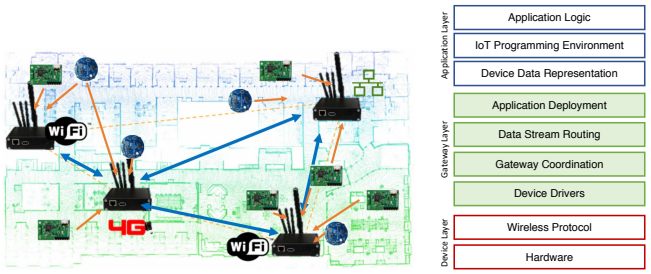
We first describe the architecture we chose for our gateway-based edge computing platform, and the rationale behind it. We then describe the design of a system to realize this architecture as well as insufficient alternatives.

A. Architectures for Edge Computing

Building on the success of cloud computing, current edge computing implementations use an architecture that largely tries to mimic cloud abstractions but runs the computation on edge servers. This centralized architecture is described in Figure 4a. AWS IoT Greengrass [10] takes this approach where the cloud sends applications to an edge server. This server interacts with all devices through gateways and executes all local applications. Just as serverless computing exploits small, short-lived pockets of unused cloud compute, Greengrass enables small *lambda* functions to leverage the capability of the edge server machine.

While conceptually attractive, mapping the centralized cloud architecture to the edge has several drawbacks. First, scaling up compute capabilities can be difficult, expensive (i.e. buying and installing a new server) [40], [41], or unsupported. Second, all data streams are centralized, incurring network traffic overhead [42]. Third, to move applications from the cloud to the edge, devices must use a standard protocol like MQTT [10], which either excludes emerging resource constrained devices or necessitates numerous gateways. Even resource optimized protocols like CoAP [43] and CBOR [44] can be difficult to map to protocols like BLE, and standardization and agreement has also remained elusive.

Instead, our proposed architecture (Figure 4b) observes that as gateways can execute applications, the edge server



(a) Platform Deployment (b) IoT Stack

Figure 5: NexusEdge platform across a floor of a building, and layers of the IoT stack for edge computing. We focus on the highlighted gateway layers, while ensuring that developments in other layers are compatible with the platform.

is largely redundant (for the set of applications that we enumerated in Section IV). Removing the server simplifies deployments and re-uses gateway hardware that is commonly already deployed. We use a fully decentralized network of gateways, which also eliminates a central point of failure. Additionally, gateways coordinate with each other to provide intelligent optimizations in data stream routing, i.e., forwarding data streams only if they are required by applications. This approach does incur overhead costs related to setup, consensus, and organization among gateways. However, this is mostly one-time, and does not affect the critical path of applications (i.e. the time taken for sensing, processing, and actuating).

B. Design Goals

To realize the selected decentralized architecture and evaluate potential design alternatives, we first describe the goals necessary for a successful design. **G1**: Manage heterogeneity. The Internet of Things is ripe with heterogeneity, encompassing different hardware platforms, data formats, communication protocols, operating modes, energy sources, and mobility patterns. Further, gateways themselves can be attached to different networks (e.g. WiFi vs. 4G). A successful design should cope with this heterogeneity. **G2**: Support applications. Ultimately the goal of any edge platform is to enable applications to leverage the available data streams. The design should provide useful APIs and abstractions for applications while reducing complexity for developers. **G3**: Minimize configuration overhead. In small deployments the configuration required to add a new device or gateway is inconsequential, however, at scale per-device configuration is laborious. A design that simplifies this enables edge computing systems to be more widely deployed. **G4**: Simplify network management. Any deployment will require management over time, and a design that aids with this management is preferable to one that does not. **G5**: Provide resiliency. As gateways are typically not as robust as server-class machines, the design should still provide resiliency for applications and data streams.

C. Design Overview

NexusEdge uses multiple gateways to provide connectivity for IoT devices deployed throughout a space, as shown in Figure 5a. Each gateway is connected to one or more IoT devices. The gateways automatically discover each other to form the gateway platform, even if the gateways are connected to different backhaul networks. Once discovered, the gateways coordinate to provide applications with the illusion that there is a single gateway with connections to all devices. Any gateway can execute applications, and the platform optimizes where to execute applications locally.

The NexusEdge platform supports IoT-on-the-edge deployments, but does not address all challenges in developing IoT systems and applications. Figure 5b provides a view of many common IoT layers, and highlights the gateway-focused aspects that NexusEdge addresses. Other layers are meant to plug into the platform, and the platform's APIs are designed to ensure compatibility as new advancements are made in other areas, including wireless protocols, device standardization, and IoT programming frameworks.

D. Gateway Discovery

NexusEdge gateways must coordinate to create a cohesive platform, and the first step is a gateway discovery process. The simplest approach would be to expect manual discovery where a user or network administrator explicitly configures a new gateway and adds it to the network. This is not consistent with **G3**, however, and complicates scaling the platform.

To avoid manual configuration, and since we assume the gateways are connected to some backhaul network (although not necessarily the wider internet), gateways could use an established network discovery protocol, such as the Simple Service Discovery Protocol (SSDP) [45] or IPv6's Neighbor Discovery Protocol (NDP) [46]. This fails in complex deployments—exactly the deployments where manual configuration is most difficult—as gateways may not be on the same LAN, and routers typically do not forward discovery messages.

Instead, we observe that since gateways by definition support wireless IoT devices that use common wireless protocols, gateways must be distributed spatially based roughly on the communication range of these protocols. Therefore, we perform wireless discovery using a standard IoT wireless protocol. Gateways send encrypted beacons and listen for other beacons to discover nearby gateways. This allows gateways to form networks based on their physical proximity, even if from a LAN networking perspective they are several hops apart.

E. Unified Gateway Platform

After gateways discover each other and create a common "gateway network", they must coordinate to provide a cohesive edge computing platform. Following the lead of today's edge-to-cloud IoT platforms like AWS IoT Greengrass [10], the platform could take a gateway-centric view and expose each gateway and the network topology to users, developers, and network administrators. This would enable the administrator to deploy applications to specific gateways and configure exactly how gateways share data streams. Since administrators know which applications they require and the physical deployment environment, they may be able to help optimize the platform.

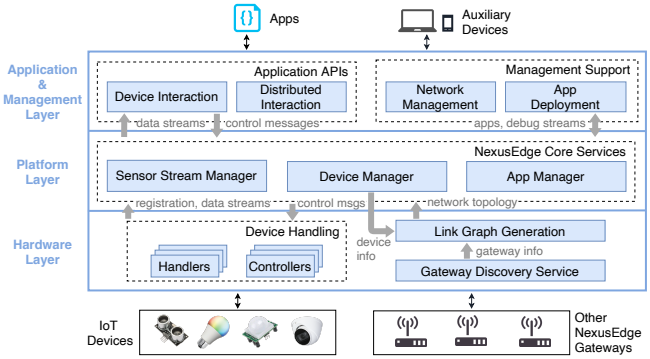


Figure 6: Layered architecture of NexusEdge.

The drawback is that a real-world deployment is likely quite complex, and it is difficult to understand the optimal placement and expected wireless communication patterns between devices and gateways. Also, this sets a high bar for using the platform, and complicates development as applications need to understand the deployment topology. In fact, existing edge computing platforms aimed at smartphones require significant overhead for edge resource discovery and usage [47], [48].

Instead, our design folds the topology complexity into the platform, and presents the abstraction of the "single gateway model", as if all devices in the deployment are connected to the same gateway. This abstraction is essential for meeting goals **G1** and **G2**. Developers write apps as if all data streams are immediately available, and the platform itself chooses which gateway to actually execute the application on and ensures the necessary data streams are available to that gateway. The tradeoff is increased routing and platform complexity, but as we show in Section VII the overhead is minimal.

To enable this abstraction, we maintain a network-wide graph data structure that encodes the topology of the gateway network with gateways as nodes, and discovery links (Section V-D) as edges. This "link graph" also tracks which devices are connected to which gateways, the backhaul addresses of each gateway, and applications that are currently running on each gateway. All algorithms for routing data, placing applications, and managing device mobility only need the link graph data structure. The gateways update and propagate changes to the link graph as the network changes over time (e.g. if a new gateway or device is added).

F. Handling Device Heterogeneity

The key task of the gateway platform after discovery and unification is to actually interface with the distributed IoT devices. However, IoT devices are inherently heterogeneous, coming from different manufacturers with different sensors, energy sources, operating modes, form factors, data formats, and wireless protocols. Addressing this heterogeneity requires answering the question: between which layers of Figure 5b should the "narrow waist" for IoT networks be put? We explore the available options.

Unifying at the wireless hardware level (e.g. Thread [49], ANT [50], DigiMesh [51], and Sigfox [52]) has gotten little traction. As wireless protocols inherently contain numerous tradeoffs, it is not clear what the one wireless standard and data format every device should agree on. Alternatively, if every device did conform to some networking standard, for example IPv6 6LoWPAN, then IoT gateways could directly support that protocol, much in the way that a WiFi access point works. If or when such a standardization occurs, the device driver layer in Figure 5b could be simplified and the rest of the gateway platform would operate as described. Again, consensus remains elusive in practice.

To avoid the overhead of full standardization, NexusEdge could specify only the data format (e.g. requiring JSON or protobufs). The gateways would need to support multiple communication protocols but device data would be in a known format with a known schema. For example, the KubeEdge [53] platform requires a device model based on YAML for each device configured on its platform [54], and Azure IoT supports an optional device format in JSON [55]. Since not all devices currently share a common schema (although efforts like One Data Model [56], Open Data Format [57], and Zigbee Alliance's Dotdot [58] are attempting to), gateways would include small "converter" software modules that re-format data from devices into its canonical format. In attempting this design, we found it unsuitable as crafting a common data format works well for a few types of sensors, but it became increasingly difficult to conform to the standard format as we added more devices to the platform.

Thus, our platform design eschews any attempt to manage the heterogeneity, and instead embraces it, as shown in the hardware layer of Figure 6. We only require that devices have some identifier so we can track them on the platform, and some general type so we can cluster them to support application APIs. Our experience indicates these requirements enable IoT scaling and broad interoperability. Data sent to or from devices is treated entirely as a binary sequence with no expectations of structure, much like the application layer data of a traditional networking packet. Interpreting data or understanding devices is left entirely to upper layers. This approach maximizes device scalability by imposing the fewest requirements on devices, and allows the design to entirely support **G1**. Note, however, this design choice does not preclude using an IoT data schema with NexusEdge, just that the platform does not enforce it. As we show in Figure 5b, NexusEdge only provides a few layers of the overall IoT stack, and other layers can be plugged in to support different deployment and application objectives.

To support this device interface, the platform includes a three-function API for managing devices. This API must be implemented for new classes of devices.

- `register(deviceId, deviceType)`: Informs the platform of a new connected device with given ID and type.
- `deliver(deviceId, data)`: Pass the specified binary data from the device to the platform. The platform ensures that it receives data only from registered devices.
- `dispatch(deviceId, data)`: Pass binary data from the platform to the specified device.

G. Auxiliary Devices

Without a central entity to manage the distributed network, there is no clear interface to manage the overall platform. To provide such an interface, the platform enables devices including laptops and smartphones to temporarily connect to the network. These devices connect to a neighboring gateway using the same discovery radio as any other gateway. This is analogous to searching for a WiFi connection from a mobile device where the device needs to be present in a WiFi router's coverage area. Once a device is connected to the platform as an auxiliary device, it can use the same internal platform APIs to manage the platform. This includes visualizing the link graph to study the network topology, deploying applications on the gateway platform, monitoring which applications are currently executing, viewing the standard logs of applications for troubleshooting, and terminating applications.

H. Application Support

The ultimate goal of the platform is to support applications that run locally on the edge operating on the rich sensor data. We do not specify the exact runtime and execution format for applications, but only specify how they interact with the core platform. Once an application is loaded (via an auxiliary device) to an arbitrary gateway, that gateway uses the link graph to identify gateways with spare compute resources, chooses a gateway which minimizes data transfer to execute the application (i.e. it tries to choose a gateway with the relevant sensors connected to it), loads the application on that gateway, and finally configures the relevant gateways to forward any relevant data streams to the executing gateway.

Due to the single gateway abstraction, developers can write applications as though data from every device in the network is immediately available. These applications interface with the platform through a narrow API with four functions. This API can be simple due to the design choice of using opaque binary data. Additional software layers outside the scope of this work can help with developers manage the binary data.

- `receive(deviceId, callback(data))`: Request data from specific devices. Due to the event-based nature of sensor data, data is returned as a callback.
- `receiveAll(deviceType, callback(data))`: Request data from all devices of a certain type.
- `send(deviceId, data)`: Send data to a specific device. The transmitted data is similarly treated as a binary sequence, and could contain an actuation command.
- `sendAll(deviceType, data)`: Send data to all devices of a certain type. For example, an application can send a new set point temperature to all thermostats.
- `poll(deviceId, request, callback(data))`: Query a specific device by sending it binary request data and expecting a response. For example, an application can request an air quality sensor to get the latest measurements.

In addition to executing an application on a single gateway, developers can also leverage the distributed nature of the platform to execute the same application on multiple gateways. The platform provides a message passing interface to let application instances interact with each other. This is especially useful to split and run applications at spatially close spaces, for example, different apartments in an apartment complex, or labs in a university building. The provided API follows.

- `disseminate(tag, data)`: Send a message to all other instances of the same application, with a tag to differentiate message types.
- `query(tag, query, callback(response))`: Retrieve disseminations for a specific tag from other instances. Similar to the data API, the events arrive as callbacks.

I. Providing Resilience

A key benefit of a distributed architecture is that the platform can provide better resilience for applications in the event of faults, when compared to a centralized architecture. Gateways can fail due to power loss, gateway movement, OS bugs, physical disturbances, or other issues. In the event of a gateway failure, the platform ensures resiliency to meet **G5**.

If a gateway that was executing an application fails, the application must be restarted on a different gateway. When the platform receives an application to schedule, it identifies an *executor* gateway to execute the application and a *watcher* gateway to watch for failures on the executor gateway. The watcher stores a copy of the application and periodically checks the link graph to see if the executor failed, and if so, the watcher chooses a new pair of executor and watcher gateways for the application, restarts the application, and stops watching. Additionally, the executor also checks for failures for its watcher, and if the watcher fails it nominates a new watcher for the application.

If a gateway that is collecting and forwarding data to an application fails, then the platform identifies one or more alternative gateways which can provide the same data streams to the application (since devices could be sending their data to multiple gateways). The executor gateway receives periodic heartbeat messages from each provider gateway, and if a provider fails, the executor uses the link graph to identify new provider gateways for the required data streams, trying to minimize the number of provider gateways (i.e., maximize the number of streams from each provider) to reduce dependency.

VI. Implementation

The implementation is open source and is available on GitHub[1].

A. Hardware and Testbed

To prototype gateways we use Raspberry Pi 4 Model B [2] boards which support onboard BLE and WiFi, and we augment with an EnOcean [59] radio via USB. Any single-board computer should be sufficient. Our testbed has BLE and EnOcean wireless devices including temperature, door, lighting, air quality, and occupancy sensors, as well as Estimote beacons [60], power monitors, and smart sockets. The gateways and devices are shown in Figure 7.

[1] https://github.com/uva-linklab/nexusedge

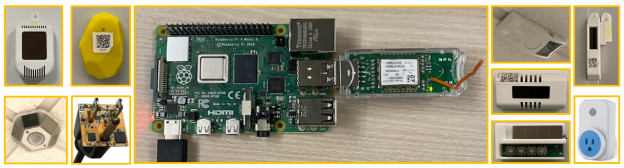


Figure 7: Gateways and devices in our deployment. Column-wise, top to bottom: CO_2 sensor, Occupancy sensor, Estimote beacon, Power meter, Raspberry Pi, Light sensor, Temperature sensor, Location beacon, Contact sensor, Smart socket.

B. Gateway Discovery using BLE

To minimize configuration overhead, gateways use their Bluetooth Low Energy (BLE) radio to discover each other. They send and receive discovery messages as BLE advertisements, using a pre-shared key that is made available during their initial configuration. The discovery messages contain the IP address of a gateway's backhaul network, and discovered gateways use this higher bandwidth network for further communication. The ubiquity of BLE in IoT gateways, its low power draw and cost, and native support for advertisements makes it a compelling choice for our discovery radio. Also, since laptops and smart phones have BLE radios, they can be *auxiliary devices* for management and application deployment.

C. Link Graph Network Abstraction

The link graph encapsulates knowledge about the current network topology. Each gateway tracks its discovered neighbors, connected devices, and running applications and encodes this as a graph. Additionally, each gateway hosts a web server which exposes this information to other gateways via well-defined endpoints. Once a gateway learns another's IP address, it can query the peer gateway to retrieve the link graph and update it with its local state. If the graph has changed, it also notifies the peer gateway. For simplicity, we encode the link graph as a JSON file. Other layers that rely on the link graph information simply parse the graph to adapt to the current network topology and structure.

D. Interfacing with Devices

NexusEdge handles device heterogeneity by not enforcing any constraints on communication protocols or data formats. It supports this with two module types: *controllers*, to communicate using a specific wired/wireless technology, and *handlers*, the device-specific code to communicate with devices. Each handler plugs into a specific controller. Handlers must implement the device-facing API defined in Section V-F. Because of a lack of standardization, each device type needs a custom handler (as in IoT systems like openHAB [61] or Home Assistant [62]), and users can load their custom controllers and handlers. The intent is a fixed number of controllers per-gateway (one for each communication channel), and device manufacturers can provide handlers for their devices.

E. Core NexusEdge Services

To enable the NexusEdge design, each gateway runs three background services, as shown in Figure 8. The *Device Manager* service loads controllers and handlers, tracks registered devices, and associates specific devices and device types with handlers. It receives data streams from registered devices, and publishes them to a *Core Service* MQTT topic for other core services to use. The *Application Manager* service accepts incoming requests to execute applications. It tracks which applications are currently running, sets up their execution environments, maintains log files, and processes termination requests. The *Sensor Stream Manager* service ensures access control of device data to applications. It branches data from the Core Service topic to different applications based on their requirements. If data is unavailable at the gateway, it requests other gateways to send data to the app.

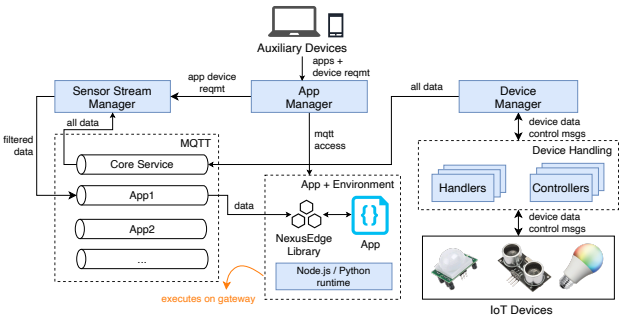


Figure 8: NexusEdge core service overview.

```
// moisture sensors: "m1", "m2", "m3", "m4", actuators: "sprinkler1"

receiveAll("moisture", function(data) { // subscribe to "moisture" sensors
  const moisture = data.values.moistureLevel;
  storeDataPoint(data.deviceId, moisture);
});

setTimeout(checkMoistureLevels, period); // periodically observe moisture

function checkMoistureLevels() {
  const minMoisture = min(sensorIds.map(id => getLatestMoistureLevel));
  if (minMoisture < threshold) // turn on sprinkler, if low moisture level
    setSprinklerState(ON_STATE);
}

function setSprinklerState(state) {
  send("sprinkler1", {"state": state});
}
```

Figure 9: Sample NexusEdge application in Node.js to control a sprinkler based on soil moisture levels.

F. Applications on NexusEdge

The application APIs described in Section V-H support arbitrary execution environments. In our prototype, we support Node.js and Python based environments, to aid in asynchronous programming and machine learning. We package the APIs as a library, using MQTT for streaming data, and HTTP for other request-based operations. The application interface uses the device-facing interface for device interaction. An example NexusEdge application is illustrated in Figure 9.

G. Deploying and Scheduling Applications on the Platform

We provide a tool on auxiliary devices (user laptops) for users to deploy applications by selecting the runtime environment and the necessary sensor streams. The application is sent to the App Manager on any NexusEdge gateway, which selects the best gateway to execute the application. Our scheduler

uses a simple heuristic based on data locality to identify the gateways that can provide the most number of required sensor streams. The scheduler also considers the CPU and memory usage of each gateway, and selects a gateway to promote locality while avoiding overburdening a single gateway.

H. Containerization and Deployability of NexusEdge

Gateway devices are heterogeneous and have different hardware or runtime environments with various subtle differences. To handle this heterogeneity, we containerize our middleware using Docker [63] to provide operating system-level virtualization. This vastly improves the ease of deployment, and abstracts out the hardware, OS, and runtime environment. The NexusEdge Docker container is public on Docker Hub [64].

Additionally, to reduce the tedious burden of deployment on many gateways, we utilize K3S [65], a lightweight version of Kubernetes [66] container orchestrator built for IoT and Edge Computing. Gateways join a cluster, and the cluster executes a DaemonSet [67] to start the NexusEdge container on all cluster nodes. DaemonSet ensures that NexusEdge starts automatically on all new gateways.

VII. Evaluation

In this section, we first evaluate certain microbenchmarks for our platform. Additionally, we focus on a case study to showcase our platform's utility for real world applications.

A. Microbenchmarks

We first compare the architecture of NexusEdge with a centralized edge server based architecture. We validate results from this experiment by comparing our platform to a real-world edge computing platform. We then evaluate the resiliency of our middleware to gateway failures. Finally, we compare the decentralization overhead for application deployment on NexusEdge to a centralized edge server.

1) Comparison with a Centralized Edge Server Architecture: To evaluate our hypothesis that a decentralized architecture adopted by NexusEdge fares better than a centralized architecture, we conduct two studies. First, we measure the CPU usage, memory usage, and the network traffic as the number of connected IoT devices increases. Next, we measure the CPU usage, memory usage, network traffic, and the application latency as the number of applications increases. We use five different Raspberry Pi devices and use them as Edge Servers and Gateways, and generate synthetic data to simulate a deployment of sensors.

Evaluation Setup: We setup a test deployment for two architectures: 1) Centralized Edge Server Architecture (ES): A single Edge Server and 4 gateways. All gateways forward data to the server. Applications are executed only on the server.

2) Decentralized NexusEdge Architecture (NE): 5 gateways operating without a server. Gateways coordinate to decide where to execute apps, and route device streams among them.

We assumed this deployment to be within the floor of a smart building, and simulated data for five classes of devices. We considered a total of 125 devices with different payload sizes, and streaming rates. We simulated 10 smart meters at 500 bytes/s, 50 temperature sensors at 100 bytes every 5 min., 50 occupancy sensors at 100 bytes every 5 min., 10 CO_2 sensors at 100 bytes every 15 min., and 5 IP cameras at 100 KB/s assuming 720p at 15 fps [68].

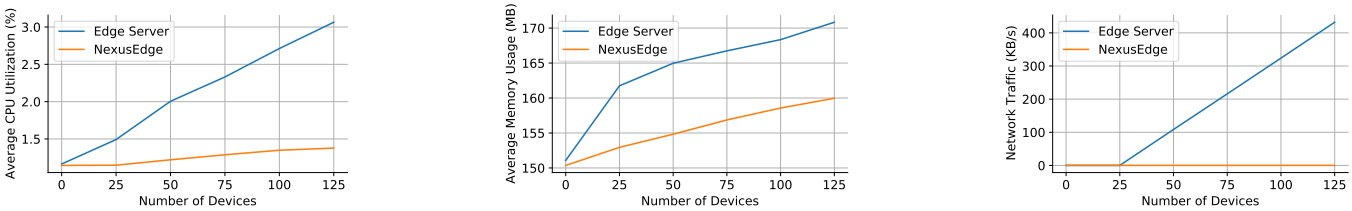
i) Evaluation of Device Scaling: For this experiment, the 125 devices are equally distributed among each edge server or gateway, i.e. each hosting 25 devices (we assume devices over a WiFi network, so even edge servers can support them). The measurements are illustrated in Figure 10. These are the key takeaways: 1) Figure 10a and Figure 10b show that resource usages are relatively higher for centralized architecture, because of getting flooded with device data. This indicates that a centralized architecture could be a bad pick if there are lots of high streaming devices. In contrast, the decentralized architecture of NexusEdge fares better in terms of resource utilization. 2) Figure 10c shows that NexusEdge has substantially lower network traffic compared to the central server, as NexusEdge gateways only stream data over the network when applications request for it.

ii) Evaluation of Application Scaling: We developed five different applications for the platform, each with its own device requirements, and deploy them in this order: 1) *Power meter anomaly detection*: Monitors power meter data from 10 smart meters, and sends an alert for anomalous power spikes. 2) *Object detection in secure areas of building*: Obtains images from five IP cameras, and uses an image classification model based on MobileNetV2 [69] to check for suspicious objects. 3) *User comfort monitor*: Collects data from 50 temperature sensors from different areas and alerts for overheating or overcooling. 4) *Air quality monitoring*: Monitors ten CO_2 sensors and alerts for dangerous ppm levels. 5) *Room scheduling*: Uses data from all occupancy sensors and user calendars to display available conference rooms.

NexusEdge schedules applications on gateways based on data locality, i.e. how devices are distributed among gateways. So we considered three different device distribution schemes in this experiment. 1) *Colocated*: All devices of a class are exclusively available on one node (we use node to refer to an edge server or a gateway). Specifically, all five IP cameras on one node, all ten CO_2 sensors on a second node, etc. 2) *Distributed*: Devices are equally distributed on all five nodes. Specifically, each node has 1 (of the 5) IP camera, 10 (of the 50) occupancy sensors, etc. 3) *Random*: Devices are randomly distributed on the five nodes.

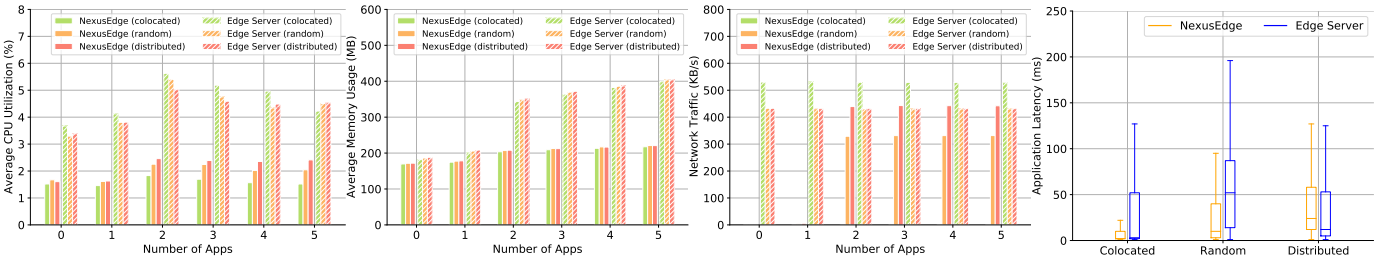
For each of these schemes, we deployed the applications successively, and measured the average CPU utilization, memory usage, network traffic, and the application latency (time it takes for an application to receive a sensor packet after it was created). The application latency does not include the machine learning inference time for the object detection application, with our intent being to measure how fast an application gets to consume data. The measurements are shown in Figure 11.

These are the key findings: 1) Average memory usage of ES is roughly 90% higher than NE, across device distributions. This can be attributed to having a single node to execute ap-



(a) CPU Utilization (b) Memory Usage (c) Network Traffic

Figure 10: Illustrates the CPU usage, memory usage, and network traffic for the device scalability study.



(a) CPU Utilization (b) Memory Usage (c) Network Traffic (d) Application Latency

Figure 11: Measurements for application scalability with varying device distribution schemes (colocated, distributed, random).

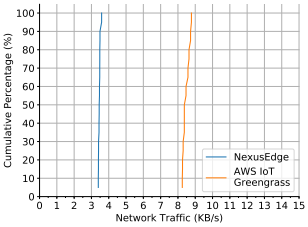


Figure 12: CDF for network traffic

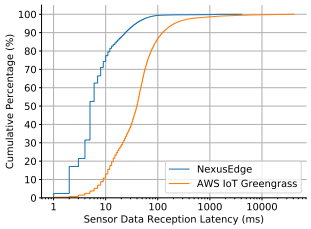


Figure 13: CDF for application latency

plications. We acknowledge that since there are more gateways used in NE as compared to ES (1 server), the total memory usage for NE would be higher than ES (1103 MB for NE, compared to 404 MB for ES, for the random device distribution). But we note however that gateways are still needed for the centralized approach (packet forwarding), and their memory use is not considered. 2) ES suffers from a higher network traffic. NE deploys applications based on the locality of devices on gateways, thereby reducing additional network traffic as shown in Figure 11. However, if devices are equally distributed among gateways, applications end up needing data from all gateways, effectively performing similar to the centralized ES architecture. 3) NexusEdge offers better application latency for the colocated and random device distributions, but is slightly higher for the distributed case (Figure 11). 4) The CPU and memory usages of gateways in NexusEdge are very low, at 3% and 250 MB (with 4 GB available), even with five apps running. This shows that gateways can indeed support edge applications, and also scales up well.

Overall, both studies highlight the benefits of NexusEdge with substantial improvement in network load, better resource usage, especially when there is spatial locality in data.

2) Comparison with AWS IoT Greengrass: To validate our results from Section VII-A1, we compare our platform with AWS IoT Greengrass. Greengrass has an architecture that is similar to the centralized edge computing architecture (Figure 4a). We evaluate with our testbed and an application for monitoring users' thermal comfort.

Evaluation Setup: This application obtains data from 70 actual sensors from our deployment (14 occupancy sensors and 56 temperature sensors). The occupancy sensors only send data when they detect motion, and the temperature sensors send data every 5 minutes. We partition the space into 14 regions based on the location of the occupancy sensors, and an app analyzes the average temperature of each region. The app periodically checks if the temperatures of regions are within an acceptable comfort range, and also reports energy wastage for unoccupied and overcooled regions. We built the application for our platform and for the Greengrass platform.

Evaluation Results: We let the application run for 30 minutes, and measured network traffic, and time taken for sensor data to reach the app. We repeat this for 10 runs and plot CDFs for the measurements, as shown in Figure 12 and Figure 13. The mean network traffic for our platform is 3.46 KB/s, 2.5x lower than the 8.48 KB/s of Greengrass. The application latency for Greengrass is 142.74 ms and 13.83 ms for NexusEdge, which is 10x faster. This is because Greengrass requires one gateway to be configured as the "Greengrass Core" which is the only one that can run apps. This limits application optimizations like shifting apps closer to device streams. We take advantage of this to reduce the latency and network traffic.

The results shown here are for a single app, and in centralized platforms it will exacerbate when there are more applications, devices, or devices with higher data rates. Increased network traffic would also result in applications that are less responsive. This experiment also highlights the benefits of executing applications at the gateway layer, one hop closer to

the devices, rather than at a central point like an edge server.

3) Resiliency to Gateway Failures: Since gateways are not as sturdy as edge servers, they can be subject to faults such as unexpected failures, deployment changes, etc. The NexusEdge middleware ensures application resiliency in two scenarios: (a) *executor failure*: if the executing gateway fails, the application is migrated to a different gateway, and (b) *provider failure*: if a gateway providing data streams to an application fails, the middleware reinstates data streams from other gateways if possible. For both scenarios, we use a periodic timer of 60 seconds to detect the failure, which is configurable. As the time taken to detect a failure depends on this period, we skip measuring the failure detection and instead evaluate the time it takes to recover once a failure is detected. We describe our evaluation setup and results below.

(a) Executor Failure: We define migration time as the time taken to restart the application on a different gateway, when the executor fails. This includes time for rescheduling the application, generating the link graph, dispatching the application to the new executor, and setting up data streams from provider gateways. Since the link graph generation time is dependent on the number of gateways and devices, we measure the migration time with varying number of gateways and devices.

We start with 2 gateways and 40 devices and go up to 5 gateways with 100 devices, with each gateway hosting 20 devices. We generate synthetic data for devices, with each device connected to *exactly* one gateway. We execute an application on one of the gateways which receives data streams from all devices. We then fail the executor gateway and measure the time taken to migrate the application once the failure is detected. We repeat this 10 times and plot the average migration time as shown in Figure 14a. Migrating an application that requires data from 100 devices distributed among 5 gateways only takes around 1.2 s, and the migration time follows a linear trend as gateways and devices increase.

(b) Provider Failure: If a gateway that was providing data streams fails, and there are other *failover gateway(s)* which can provide the same data streams, the middleware requests the failover gateway(s) for the data. We define the recovery time as the time it takes to complete these requests and reinstate data streams. This includes generating the link graph, identifying failover gateways, and requesting them for data.

Similar to the previous experiment, we varied the number of devices and gateways, but from 3 gateways and 60 devices to 5 gateways and 100 devices. We excluded the 2 gateways case, since there is no failover gateway when one of the gateway fails. For 4 gateways, after a providing gateway fails, there can be 1 or 2 failover gateways that can provide the streams. As the number of failover gateways increase, the recovery time slightly increases as more gateways need to be requested to forward data streams. We distributed devices among gateways so that there is some redundant providers and thus a scope to recover when the original provider gateway fails.

We induced failure for one of the provider gateways, and measured the average recovery time for 10 runs as shown in Figure 14b. Since reinstating data streams doesn't require migrating the application, the recovery time is faster than migration time, taking only 712.3 ms even with 5 gateways, 100 devices while using 3 failover gateways.

Both these experiments show that NexusEdge can recover quickly from gateway failures, by migrating applications or reinstating data streams from alternative providers. It highlights the significance of our middleware in building a cohesive edge platform with gateways, that can operate with resiliency without suffering from a single point of failure.

4) Decentralization Overhead: Shifting from an edge server to decentralized gateways adds some overhead for gateway coordination, and we evaluate if this overhead is reasonable. We compare the time taken to deploy an application on a centralized edge server and on NexusEdge. For the edge server, scheduling is not needed as the application executes on the same machine. Also, since all device streams are available on the server, deploying the application only includes sending it to the server, and to subscribe to the MQTT broker providing all streams. However, for NexusEdge, deploying an application involves sending it to a scheduling gateway, generating the link graph, scheduling the application, dispatching the application to an executing gateway, setting up streams from provider gateways, and subscribing to all data streams.

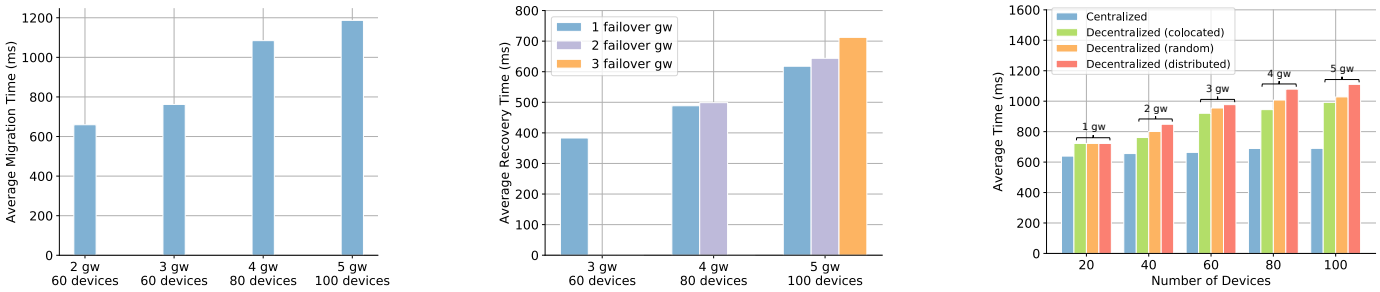
We deploy an application which receives data from multiple devices (virtual sensors sending a small payload every 5s), and measure the deployment time at varying number of devices. For centralized, the application always executes on the server and we vary the number of devices from 20 to 100, at steps of 20. For decentralized, we vary the numbers of gateways and devices starting from 1 gateway with 20 devices and going up to 5 gateways with 100 devices. We also consider three device distribution schemes i.e., how devices are distributed among the gateways: colocated (best case), random (average case), and fully distributed (worst case), as the distribution affects the latency (Section VII-A1). The measurements from our study are shown in Figure 14c.

As expected, the deployment time for centralized is lower than decentralized. But even with 5 gateways, 100 devices, and the worst case device distribution, the deployment time for NexusEdge is 1110.4ms compared to 690.1ms for centralized, with an overhead of 420ms. This overhead is reasonable since deployment for long running applications happens infrequently. Also, applications in centralized require additional time to filter out data streams, which is handled for NexusEdge and included in its deployment time.

B. Case Study: Federated Learning without Cloud Support

We describe an illustrative example application on NexusEdge to predict turn on times of an appliance based on its power data. Accuracy of the application can be improved by training with data of the same appliance type from different sources. For instance, usage data of dryers from multiple users can be used, but users may not be comfortable sharing such data. So we modelled this as a federated learning problem.

Federated ML is typically used by mobile devices to learn a shared prediction model. Devices keep their data private, but



(a) Resilience to Executor Failures (b) Resilience to Provider Failures (c) Decentralization Overhead

Figure 14: a) and b) illustrate the application migration and recovery time respectively for the middleware when gateways fail. c) illustrates the application deployment time on NexusEdge compared to a centralized edge server.

can still run predictions, and retrain their models. Changes from all local models are sent to the cloud to improve the shared model, which is then synchronized back. We demonstrate how our platform can natively run federated ML on gateway devices without the need of a cloud.

1) Modeling Federated ML on NexusEdge: As described in Section V-H, NexusEdge allows an application to be distributed on different gateways. Each app instance uses the `receive` function to receive power meter data from the gateway it runs on. The apps are initialized with the global model, and retrain their models whenever they get substantial amount of data. Over time, when the data received by an app crosses a threshold, it uses the `query` function to request model parameters from all other apps. It then performs a federation step by taking a weighted average of model weights to generate an updated model. It uses the `disseminate` function to share this updated model to all other apps.

Measuring Overhead and Performance: The query and disseminate calls only amount to 25 LOC. The model parameter exchanges were each of 8 KB. One federation step required 32 KB of data transfer between the apps. For the federated learning model, we used a neural network with two hidden layers and used all time-based features. We simulated appliance level power data from the UK-DALE dataset [70] (around 900,000 data points). The model's accuracy increased from 73.39% to 78.93% after one round of federated learning. To measure the effectiveness of the distributed approach, we also compared this with a non-federated baseline (which has access to all data) which reported an accuracy of 79.93%.

The accuracy numbers do not mean much in the context of our platform. The key takeaway is that NexusEdge can ease the burden in developing complex IoT applications by providing useful abstractions, including a distributed interaction API.

VIII. Discussion

We list interesting research problems that can be studied with our platform, and also study limitations of our work.

User-Driven Privacy Enforcement in IoT: Device data collected from environments directly affect the privacy of people in those spaces. However, users have no way of controlling which applications can access their data. NexusEdge's access control can be extended to allow users to set custom privacy policies, for example, permitting access to their occupancy data only during work hours.

Matching Applications with Gateway Capabilities: Gateways are becoming increasingly diverse, with powerful GPUs [71], multi-protocol support [72], and custom accelerators [73]. But, this diversity is not considered by platforms when mapping apps onto the edge. A new scheduler can match app requirements to gateway capabilities, for example, running a machine learning app on a gateway with GPUs, even if the gateway is not close to the sensors.

Security Challenges: Security is a major consideration for edge computing systems. We use a limited threat model in this work with preliminary security provisions. To enhance security, we need to explore different threat models for the platform, apps, devices, auxiliary devices, and data. Certain interfaces including device registration, gateway discovery, and auxiliary device connections, could support security features.

IX. Conclusion

Edge computing platforms generally consider the IoT gateway layer an implementation detail, yet how data flows from devices to edge resources affects performance, robustness, and scalability. Further, this layer presents significant untapped potential for addressing key challenges in IoT deployments: latency, reliability, and privacy. NexusEdge utilizes a distributed architecture for the gateway layer, and provides a concrete example of the abstractions, design decisions, and software components required to realize it. Through microbenchmarks, a real-world deployment, and a case study, we demonstrate how NexusEdge increases performance compared to a centralized architecture. However, NexusEdge still remains one new component of an otherwise complex and nuanced IoT computing platform, and we show how NexusEdge is extensible to support related techniques that address other IoT challenges. As the IoT is deployed for new use cases and managing scale is a growing burden, a decentralized gateway layer promotes scalability while leveraging hardware that is already deployed.

Acknowledgment

This work was supported by the University of Virginia Strategic Investment Fund under grant SIF128, and by the National Science Foundation under award CNS-2144940.

REFERENCES

[1] L. Dignan. (2019) Iot devices to generate 79.4zb of data in 2025, says idc. [Online]. Available: https://www.zdnet.com/article/iot-devices-to-generate-79-4zb-of-data-in-2025-says-idc/

[2] T. R. P. Foundation. (2021) Raspberry pi 4. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b

[3] P. Bellavista and A. Zanni, "Feasibility of fog computing deployment based on docker containerization over raspberrypi," in *Proceedings of the 18th international conference on distributed computing and networking*, 2017, pp. 1–10.

[4] F. Jalali, A. Vishwanath, J. De Hoog, and F. Suits, "Interconnecting fog computing and microgrids for greening iot," in *2016 IEEE Innovative Smart Grid Technologies-Asia (ISGT-Asia)*. IEEE, 2016, pp. 693–698.

[5] J. Canedo and A. Skjellum, "Using machine learning to secure iot systems," in *2016 14th annual conference on privacy, security and trust (PST)*. IEEE, 2016, pp. 219–222.

[6] S. Cirani, G. Ferrari, N. Iotti, and M. Picone, "The iot hub: A fog node for seamless management of heterogeneous connected smart objects," in *2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking-Workshops (SECON Workshops)*. IEEE, 2015, pp. 1–6.

[7] A. W. Services. (2020) Developer guide - aws iot greengrass. [Online]. Available: https://docs.aws.amazon.com/greengrass/latest/developerguide/setup-filter.rpi.html

[8] Microsoft. (2020) Developer guide - azure iot hub. [Online]. Available: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-raspberry-pi-kit-node-get-started

[9] L. Zhou and H. Guo, "Anomaly detection methods for iiot networks," in *2018 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*. IEEE, 2018, pp. 214–219.

[10] A. W. Services. (2020) Aws iot greengrass. [Online]. Available: https://aws.amazon.com/greengrass/

[11] J. Clemente, M. Valero, J. Mohammadpour, X. Li, and W. Song, "Fog computing middleware for distributed cooperative data analytics," in *2017 IEEE Fog World Congress (FWC)*. IEEE, 2017, pp. 1–6.

[12] B.-Y. Ooi, Z.-W. Kong, W.-K. Lee, S.-Y. Liew, and S. Shirmohammadi, "A collaborative iot-gateway architecture for reliable and cost effective measurements," *IEEE Instrumentation & Measurement Magazine*, vol. 22, no. 6, pp. 11–17, 2019.

[13] Microsoft. (2020) Azure iot edge. [Online]. Available: https://azure.microsoft.com/en-us/services/iot-edge/

[14] G. Guan, B. Li, Y. Gao, Y. Zhang, J. Bu, and W. Dong, "Tinylink 2.0: integrating device, cloud, and client development for iot applications," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020, pp. 1–13.

[15] R. Yus, G. Bouloukakis, S. Mehrotra, and N. Venkatasubramanian, "Abstracting interactions with iot devices towards a semantic vision of smart spaces," in *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, 2019, pp. 91–100.

[16] J. Gascon-Samson, M. Rafiuzzaman, and K. Pattabiraman, "Thingsjs: Towards a flexible and self-adaptable middleware for dynamic and heterogeneous iot environments," in *Proceedings of the 4th Workshop on Middleware and Applications for the Internet of Things*, 2017, pp. 11–16.

[17] A. Essameldin, M. N. Hoque, and K. A. Harras, "More than the sum of its things: Resource sharing across iots at the edge," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2020, pp. 206–219.

[18] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *computer*, vol. 50, no. 10, pp. 58–67, 2017.

[19] H. Wang and J. Xie, "You can enjoy augmented reality while running around: An edge-based mobile ar system," in *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2021, pp. 381–385.

[20] G. Tricomi, Z. Benomar, F. Aragona, G. Merlino, F. Longo, and A. Puliafito, "A nodered-based dashboard to deploy pipelines on top of iot infrastructure," in *2020 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2020, pp. 122–129.

[21] R. Tips. (2022) What's the best micro sd card for raspberry pi? (benchmark). [Online]. Available: https://raspberrytips.com/best-sd-card-raspberry-pi/

[22] IFTTT. (2021) Activate a powerview shade scene when the temperature changes. [Online]. Available: https://ifttt.com/applets/QxHysZRg

[23] I. Hue and August. (2021) Automatically turn on your hue lights on when you unlock your august smart lock. [Online]. Available: https://ifttt.com/applets/mqs4CUv9

[24] D. A. Winkler and A. E. Cerpa, "Wisdom: watering intelligently at scale with distributed optimization and modeling," in *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, 2019, pp. 219–231.

[25] T. Meyer and G. Hancke, "Design of a smart sprinkler system," in *TENCON 2015-2015 IEEE Region 10 Conference*. IEEE, 2015, pp. 1–6.

[26] G. R. Newsham and B. J. Birt, "Building-level occupancy data to improve arima-based electricity use forecasts," in *Proceedings of the 2nd ACM workshop on embedded sensing systems for energy-efficiency in building*, 2010, pp. 13–18.

[27] Y. Tang, S. Zhao, C.-W. Ten, K. Zhang, and T. Logenthiran, "Establishment of enhanced load modeling by correlating with occupancy information," *IEEE Transactions on Smart Grid*, vol. 11, no. 2, pp. 1702–1713, 2019.

[28] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 kb ram for the internet of things," in *International Conference on Machine Learning*. PMLR, 2017, pp. 1935–1944.

[29] S. S. Ogden and T. Guo, "{MODI}: Mobile deep inference made efficient by edge computing," in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[30] T.-H. Yang, H.-Y. Cheng, C.-L. Yang, I.-C. Tseng, H.-W. Hu, H.-S. Chang, and H.-P. Li, "Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 236–249.

[31] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[32] C. Gupta, A. S. Suggala, A. Goyal, H. V. Simhadri, B. Paranjape, A. Kumar, S. Goyal, R. Udupa, M. Varma, and P. Jain, "Protonn: Compressed and accurate knn for resource-scarce devices," in *International Conference on Machine Learning*. PMLR, 2017, pp. 1331–1340.

[33] C. C.-H. Hsu, M. Y.-C. Wang, H. C. Shen, R. H.-C. Chiang, and C. H. Wen, "Fallcare+: An iot surveillance system for fall detection," in *2017 International conference on applied system innovation (ICASI)*. IEEE, 2017, pp. 921–922.

[34] S. Tuli, N. Basumatary, and R. Buyya, "Edgelens: Deep learning based object detection in integrated iot, fog and cloud computing environments," in *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*. IEEE, 2019, pp. 496–502.

[35] G. Codeluppi, A. Cilfone, L. Davoli, and G. Ferrari, "Ai at the edge: A smart gateway for greenhouse air temperature forecasting," in *2020 IEEE International Workshop on Metrology for Agriculture and Forestry (MetroAgriFor)*. IEEE, 2020, pp. 348–353.

[36] S. R. Hussain, S. Mehnaz, S. Nirjon, and E. Bertino, "Seamblue: Seamless bluetooth low energy connection migration for unmodified iot devices," in *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks*, ser. EWSN ’17. USA: Junction Publishing, 2017, pp. 132–143. [Online]. Available: http://dl.acm.org/citation.cfm?id=3108009.3108027

[37] T. Beke, E. Dijk, T. Ozcelebi, and R. Verhoeven, "Time synchronization in iot mesh networks," in *2020 International Symposium on Networks, Computers and Communications (ISNCC)*. IEEE, 2020, pp. 1–8.

[38] S. Choi and J.-H. Lee, "Blockchain-based distributed firmware update architecture for iot devices," *IEEE Access*, vol. 8, pp. 37 518–37 525, 2020.

[39] M. Banerjee, C. Borges, K.-K. R. Choo, J. Lee, and C. Nicopoulos, "A hardware-assisted heartbeat mechanism for fault identification in large-scale iot systems," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[40] F. Wang, X. Huang, H. Nian, Q. He, Y. Yang, and C. Zhang, "Cost-effective edge server placement in edge computing," in *Proceedings of the 2019 5th International Conference on Systems, Control and Communications*, ser. ICSCC 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 6–10. [Online]. Available: https://doi.org/10.1145/3377458.3377461

[41] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE*

Communications Surveys Tutorials, vol. 19, no. 4, pp. 2322–2358, Fourthquarter 2017.

[42] J. Ren, D. Zhang, S. He, Y. Zhang, and T. Li, "A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet," *ACM Comput. Surv.*, vol. 52, no. 6, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3362031

[43] IETF. (2014) The constrained application protocol (coap). [Online]. Available: https://tools.ietf.org/html/rfc7252

[44] I. E. T. Force. (2020) Rfc 8949 concise binary object representation. [Online]. Available: https://cbor.io/

[45] S. Albright, P. J. Leach, Y. Gu, Y. Y. Goland, and T. Cai, "Simple Service Discovery Protocol/1.0," Internet Engineering Task Force, Internet-Draft draft-cai-ssdp-v1-03, Nov. 1999, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-cai-ssdp-v1-03

[46] W. A. Simpson, D. T. Narten, E. Nordmark, and H. Soliman, "Neighbor Discovery for IP version 6 (IPv6)," RFC 4861, Sep. 2007. [Online]. Available: https://www.rfc-editor.org/info/rfc4861

[47] I. Murturi, C. Avasalcai, C. Tsigkanos, and S. Dustdar, "Edge-to-edge resource discovery using metadata replication," in *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, 2019, pp. 1–6.

[48] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, 2016, pp. 20–26.

[49] T. Group. (2021) Thread protocol. [Online]. Available: https://www.threadgroup.org/

[50] G. C. Inc. (2021) Ant / ant+ defined. [Online]. Available: https://www.thisisant.com/developer/ant-plus/ant-antplus-defined

[51] DigiMesh. (2021) Digimesh products. [Online]. Available: https://www.digi.com/products/browse/digimesh

[52] S. S.A. (2021) Sigfox - the global communications service provider for the internet of things (iot). [Online]. Available: https://www.sigfox.com/

[53] C. N. C. Foundation. (2020) Kubeedge. [Online]. Available: https://kubeedge.io/en/

[54] K. Project. (2021) Device model for kubeedge. [Online]. Available: https://kubeedge.io/en/docs/developer/device_crd/

[55] Microsoft. (2021) Understand and use device twins in iot hub. [Online]. Available: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-device-twins

[56] O. D. Model. (2021) Onedm: Solving the problem of lack of a common data model for iot and iot devices. [Online]. Available: https://onedm.org/

[57] T. O. Group. (2021) Open data format (o-df), an open group internet of things (iot) standard. [Online]. Available: http://www.opengroup.org/iot/odf/

[58] Z. Alliance. (2021) Dotdot: a common language for the smart objects. [Online]. Available: https://zigbeealliance.org/solution/dotdot/

[59] E. GmbH. (2019) Energy harvesting wireless sensor solutions and networks from enocean. [Online]. Available: https://www.enocean.com/

[60] Estimote. (2021) Estimote proximity beacons. [Online]. Available: https://estimote.com/

[61] openHAB Community. (2021) openhab. [Online]. Available: https://www.openhab.org/

[62] H. Assistant. (2021) Open source home assistant. [Online]. Available: https://www.home-assistant.io/

[63] D. Inc. (2022) Docker. [Online]. Available: https://www.docker.com/

[64] DockerHub. (2022) Docker hub. [Online]. Available: https://www.docker.com/products/docker-hub/

[65] K. P. Authors. (2022) Lightweight kubernetes. [Online]. Available: https://k3s.io/

[66] T. K. Authors. (2022) Kubernetes - production-grade container orchestration. [Online]. Available: https://kubernetes.io/

[67] DaemonSet. (2022) Kubernetes daemonset. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

[68] Arxys. (2021) Nvr bandwidth and storage calculator. [Online]. Available: https://www.arxys.com/bandwidth-storage-calculator/

[69] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[70] J. Kelly and W. Knottenbelt, "The UK-DALE dataset, domestic appliance-level electricity demand and whole-house demand from five UK homes," *Scientific Data*, vol. 2, no. 150007, 2015.

[71] N. Corporation. (2021) Jetson xavier nx developer kit. [Online]. Available: https://developer.nvidia.com/embedded/jetson-xavier-nx-devkit

[72] L. Intwine Connect. (2021) Intwine connected gateway. [Online]. Available: https://www.intwineconnect.com/index.php?p=products-and-platforms/intwine-m2m-enablement-kit

[73] P. Gonzalez-Guerrero, T. Tracy II, X. Guo, R. Sreekumar, M. Lenjani, K. Skadron, and M. R. Stan, "Towards on-node machine learning for ultra-low-power sensors using asynchronous σ δ streams," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 16, no. 4, pp. 1–20, 2020.