Leveraging Dataflow as an Intermediate Representation for Portable Edge Deployments

Rich Wolski
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, USA

Chandra Krintz
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, USA

Markus Mock
Department of Computer Science
HAW Landshut
Landshut, Germany

***Abstract*—In this paper, we investigate how to leverage the dataflow programming model to facilitate the deployment of multi-scale (sensors-edge-cloud) IoT applications end-to-end. Today, IoT deployment management is error-prone, tedious, and manual – yet deployment configuration can have a significant effect on application energy consumption and performance. Our approach, called deployment-as-code, provides programming directives that distribute application components across heterogeneous devices (including microcontrollers), aid management of the application lifecycle and facilitate optimization in ways not available from traditional programming systems. We implement our approach by extending an open-source dataflow programming system for IoT, which we use to evaluate the energy consumption of different deployment configurations and IoT applications.**

Keywords: IoT, dataflow, serverless, FaaS, embedded systems

I. Introduction

The Internet of Things (IoT) is a computing fabric that enables ordinary, physical objects to monitor, analyze, actuate, and control their environment automatically. This fabric consists of sensing and computing devices that are embedded in the physical environment, that employ a range of energy sources, and that interoperate with co-located devices and remote cloud services via one or more communication networks (if/when available). The heterogeneous, geographically distributed, and failure-prone nature of these deployments require that developers design their applications to be both robust to failures and *multiscale*, i.e., able to leverage the battery-powered or resource-constrained sensors, single-board computers, mobile devices, and public/private clouds that comprise the edge-cloud continuum.

To enable this, new programming platforms have emerged that attempt to simplify the development of IoT applications, often leveraging what has become known as the serverless computing paradigm[1]. These platforms combine device-specific software development kits (SDKs) with network protocol stacks to interface devices to "Functions-as-a-Service" (FaaS) platforms that are either edge or cloud hosted. Serverless computing abstracts away the complexities introduced by deployment distribution, system management, networking configuration, etc. On FaaS systems, programmers implement their applications by writing simple event handlers (i.e., program functions), which they register with the FaaS platform for invocation when an event "fires." Application functions are stateless, use integrated services for persistence and coordination, are ephemeral (execute only when fired) and are easily composed into microservices applications [2], [3].

Because most FaaS IoT platforms today derive from successful cloud technologies, they are difficult to use in some IoT settings. In particular, because the cloud is designed to provide a virtualized "always on" set of services (e.g., as a computing utility), these derivatives often lack features and capabilities necessary to implement effective distributed IoT deployments, particularly in remote, outdoor settings where network and power infrastructure may be intermittently and unpredictably available. Moreover, because they were designed for resource-rich systems, most do not support sensors and devices with severe resource constraints or non-traditional operating systems uniformly. Independent of the platform, FaaS's inherent event driven (i.e., asynchronous, highly concurrent) nature can also make application development challenging to reason about.

The LAMINAR programming system has been proposed recently to address these challenges [4]. LAMINAR overlays a dataflow programming model over a fault-resilient, log-based FaaS runtime. Dataflow is a programming paradigm that formulates programs as directed graphs in which nodes are data-parallel functions and edges represent the flow of data between them. Dataflow is uniquely amenable to IoT programming since it focuses on data instead of control flow, which is useful for stream processing, event-driven computing, and highly parallel and concurrent workloads (all characteristics of modern IoT applications). Moreover, dataflow has been used successfully for asynchronous and parallel systems [5], [6], [7], and a wide range of data analytics [8], [9], [10] to simplify data processing at scale.

In this paper, we investigate how to leverage the combination of dataflow programming and log-based runtime to simplify IoT application *deployment* – the assignment, commissioning, and configuration of application components across the devices in a distributed IoT setting – in addition to distributed execution. Given the multiscale capabilities of IoT, deployment can have a dramatic effect on functional and non-functional (e.g., failure resilience, energy efficiency, etc.) application behavior. Today, such management is error-prone, tedious, and manual. Our approach, called *deployment-as-code*, provides directives that distribute application components across multiscale devices in an IoT deployment,

aid management of the application lifecycle, and facilitate optimization in ways not possible with traditional FaaS/asynchronous systems. Key to this approach is that the deployment directives are, themselves, single-assignment operations that are part of the dataflow programming environment.

To unify deployment and execution, we build upon LAMINAR with these key contributions

- We extend LAMINAR with end-to-end support for microcontrollers and sensor networks and leverage its preamble mechanism to enable programmatic energy-aware deployment management.
- We develop an alternative messaging layer for LAMINAR that is based on MQTT [11] – a popular publish/subscribe protocol used in many IoT contexts.
- We extend the LAMINAR preamble so that devices can be programmed, and their deployments can be configured, uniformly using the dataflow programming model.

Microcontroller devices play a critical role in most IoT systems and using a single unifying programming model for these and more resource-rich devices has the potential to significantly improve IoT programmer productivity. Our contributions show how application deployment spans device scales ranging from microcontrollers to the cloud can be developed as part of the application development process itself, using dataflow semantics.

We illustrate these results empirically through a study of the energy requirements associated with different LAMINAR configurations using end-to-end IoT applications. Our results indicate that deployment configuration has a significant effect on energy efficiency for some edge devices, and predicting the most efficient deployment is difficult.

In the following sections, we first overview related work and the LAMINAR programming system that we extend and evaluate in this work. We describe how we use the LAMINAR abstractions to facilitate a "deployment-as-code" IoT deployment management methodology. We then detail our experimental methodology and empirical evaluation. We measure the energy consumption of different deployment scenarios using a real IoT application and demonstrate how this approach improves programmer productivity and energy-efficient application configuration. We plan to make our extensions available as open source when/if this paper is accepted.

II. BACKGROUND AND RELATED WORK

Our work leverages language-level abstractions and dataflow programming model semantics to automate energy-aware IoT application deployment. Although dataflow programming has been widely studied for decades [12], [13], [14], [15], it has received renewed interest recently for large-scale data analytics [8], [9], [10]. These advances provide a simple programming model for large-scale, parallel processing of structured and semi-structured data on commodity clusters or cloud servers [8], [9], [10]. Their execution engines automatically schedule, place, synchronize, and manage faults for these workloads. MapReduce [8] represents programs as a bipartite graph and Dryad [9] uses a more general directed acyclic graph (DAG) (like LAMINAR).

These systems have been extended in multiple ways to reduce their restrictions and support a broader range of algorithms with greater efficiency [16], [17]. Ciel [10] extends these programming models with better support for iterative computations. Specifically, it adds dynamic control flow creation while maintaining fault resiliency. Although Ciel is more dynamic, the static specification of a fault-resilient deployment (nodes can come and go) works well for IoT applications. Unfortunately, since these systems were designed for resource-rich systems, they do not work in multiscale or wide-area settings. Moreover, deployment is straightforward because cluster/cloud systems are significantly more homogeneous than IoT settings.

A. LAMINAR *Programming System*

The open-source LAMINAR programming system was recently proposed to make use of the dataflow programming model for IoT applications that run over heterogeneous, fault-prone distributed systems [4]. LAMINAR augments the applicative or functional programming semantics of dataflow [18] with support for distributed deployment, partial failures, and crash recovery. Applicative semantics enable application robustness through idempotency. Moreover, all computations are functional so that any computation can be "replayed" to implement "at-least-once" QoS semantics.

LAMINAR integrates a log-based, multi-scale, FaaS runtime [19]. Functions are stateless and persist program state via network-transparent logs that are append-only and lock-free. Logs are less complex and more resilient than their file system and database counterparts. Logs also enable causal tracking and failure recovery [19], [20], [21], [22]. Thus, a program that uses logs exclusively as program variable storage is single-assignment and, thus, functional. Logs are named using URN's with no log spanning more than a single host. The runtime implements triggered function invocation in response to data being appended to a log (i.e., event handlers where the only event is a log-append). The runtime uses ZeroMQ over POSIX for log appends across devices. Although it is difficult to compare LAMINAR with a semi-equivalent C version that is not crash-consistent and uses locks, LAMINAR's performance ranges between in-memory C and C with simple Linux file-based persistence [4].

A LAMINAR program consists of four primary program constructs. *Operands* are external computations that introduce data to an application. These include sensor readings (in an IoT context), database reads, remote API calls, or arbitrary program functions from a program or script capable of exercising LAMINAR's API. *Nodes* perform computations on data using stateless functions written by the programmer in C/C++. *Edges* express data flow between nodes. *Graphs* implement program scoping and modularity. A program graph contains nodes and edges organized as a directed, acyclic graph (DAG). Operands carry the graph's initial inputs (e.g., the inputs to the program), and all node outputs are available at program termination.

Nodes "fire" (execute their computation) when all their inputs are available. Nodes can be composed in a hierarchy where an enclosing graph treats a subgraph as a single node. However, graphs in LAMINAR are not strict with respect to graph boundaries. Nodes in a graph can trigger dependent nodes in other graphs without waiting for all nodes in the graph to complete. This lack of graph strictness facilitates maximal parallelism, which can significantly benefit performance for multicore systems.

III. DEPLOYMENT-AS-CODE (DAC)

In this work, we propose "deployment-as-code (DAC)" – the expression of deployment directives that distribute application components to machines and that manage the application's overall lifecycle. DAC is similar in spirit to infrastructure-as-code [23] systems like Kubernetes and Terraform but designed for the most resource-constrained end of the edge-to-cloud continuum. Our approach takes advantage of dataflow semantics and program structure exposed by LAMINAR (e.g., the program graph, mapping of nodes to physical hosts, log placement, etc.) to enable developers to program the deployment of their applications and enable the system to optimize this deployment, e.g., to reduce energy consumption. Specifically, we perform graph embedding of execution directives via a simple API, as part of the dataflow specification for a program. By doing so, we enable (i) the compilation process to automatically construct per-device executable images (consisting of both the deployment context and the program code), and (ii) the dataflow runtime to automate ingress and egress of data as well as computation scheduling and execution order for the application.

To enable this, we extend the LAMINAR build process and its deployment configuration mechanisms (the LAMINAR preamble). To show that this approach generalizes to devices common to edge/fog deployments, we also extend LAMINAR with support for microcontrollers and sensor networks. The result is a software system that enables heterogeneous devices to be programmed and deployed uniformly, enabling deployment modifications to be affected with minimal code changes and developer effort. We use this approach to explore the energy consumption of different IoT application deployment configurations (since it is critical for battery-powered edge/fog devices). Other studies have shown that LAMINAR is also efficient in terms of latency and end-to-end performance [4].

To test the effectiveness of DAC, we introduce a set of extensions to LAMINAR that provide support for non-Linux and less capable devices (e.g., microcontrollers, sensors), as well as mobile devices that may not have symmetric network connectivity with the other machines or platforms in a deployment. In particular, the LAMINAR messaging layer assumes that network connections can be initiated from any node. In many deployments, however, only connections from "inside" the firewall or private/sensor network can be initiated.

To support asymmetric connectivity and more general deployability, we extend the LAMINAR messaging interface with support for MQTT as a configuration option. Supporting MQTT transport enables a wide range of heterogeneous devices to be integrated into a LAMINAR IoT deployment with minimal porting effort. Moreover, MQTT is sufficiently efficient when used for simple messaging, and popular IoT/sensor network protocols support it (e.g., Zigbee, Z-Wave, LoRaWAN, as well as cellular IoT and vehicular networks).

Our extensions require that each runtime instance in a deployment establish a network connection with an MQTT broker that is outside all firewalls and sensor networks. We provide a message-level transparent gateway that translates MQTT messages to ZeroMQ messages recognized by the runtime and vice versa. Each runtime instance using the MQTT transport requires its own transparent gateway, but gateways can share one or more MQTT brokers. We perform per-message authorization and password authentication to MQTT brokers in a deployment by extending LAMINAR with CAPLets [24] (capability-based, fine-grained access control).

Also, as part of extending LAMINAR for this study, we introduce two new configuration options for implementing LAMINAR logs: a file-based option that uses either the POSIX file system (where available) or LittleFS [25], where POSIX is not supported. The current LAMINAR log implementation relies on Linux memory-mapped files, which are not typically available on microcontroller-based or mobile devices. We also add an in-memory-only implementation of runtime logs without persistence using in-memory data structures to compare it to the performance of flash-memory persistence (using LittleFS or POSIX) performance and for devices with no access to persistent storage.

A. Programming Deployments

The key insight of DAC is that the dataflow programming model can be used to encode both the application and the deployment – the assignment of computations to execution sites and communication events to network transits – in a uniform way. All dataflow programming systems must, at some level, encode an assignment of computations to processors and messages between computations to networks or other communication substrates. This mapping can be delayed until runtime (e.g., using a "work-stealing" scheduler in which idle nodes acquire enabled computations from a pool of computations waiting to execute) or at compile time by embedding the program DAG into a graph representing the deployment topology (i.e., computational sites and network connectivity).

Our approach combines compile-time embedding with runtime support to enable deployment across different device scales and capabilities. The key insight is that the dataflow representation (i.e., the program DAG) can be used as the portable intermediate representation of the application – bringing write-once-run-anywhere [26] to heterogeneous edge and fog deployments. However, how devices are "commissioned" (added to a deployment so they may be used to execute a specific application component) varies widely by device type. In particular, embedded microcontroller devices often require cross-compiled code containing the operating system and runtime environment along with the application to be

"uploaded" as a complete image and the device restarted to implement commissioning.

Using LAMINAR, developers encode a program as a directed, acyclic graph (DAG) in a preamble section of the source code. This preamble must execute before application computations. The preamble initializes the data structures used by the LAMINAR runtime and records the node-to-host mappings in LAMINAR logs. While the dataflow program assigned to each node can differ, the preamble preceding the application code must be the same in all code that comprises a deployment. Note that the preamble and application dataflow codes are part of the same source code compiled for each node and then commissioned on that node.

We extend the LAMINAR preamble abstraction to allow it to be decoupled from the application source. The DAG specification API used in the preamble encodes nodes (add_node), program inputs (add_operand), and edges between nodes (subscribe). Nodes are wrappers for application functions. To encode a deployment, developers specify physical hosts (add_host) each with a unique integer identifier that must be included in the add_host specifiers to indicate a node-to-host assignment. Each host in a deployment must include a set_host specifier in its preamble to establish its identifier.

Figures 1 and Listing 1 show the DAG and preamble for an example application, which we describe in the next section. For DAC, we extract the preamble out into a separately compiled code component. The preamble is compiled into an executable binary on systems with full operating system process support. For microcontrollers, it is loaded as part of image creation and executed as part of the "setup" routine.

This separation offers two benefits. First, a developer can change deployments without recoding or recompiling the application program. The preamble creates LAMINAR logs that the runtime uses to execute the application (termed the application "body" in our parlance). A change in deployment changes only these logs and not the application code itself. Secondly, the LAMINAR logs are binary compatible and self-describing across storage implementations. Thus, it is possible to create the preamble logs on one host and distribute them (via a network copy) to all hosts in a deployment using the same storage technology.

Note that network topology information is not specified in the preamble (the representation is network-transparent). Thus, we assume that the deployment engineer understands the underlying network connectivity topology and will make node-to-host assignments such that any pair of nodes in the program graph connected by a directed edge will be assigned to hosts that can communicate. We are developing support for checking this at deployment time as part of future work.

Our approach also obviates the need for distributed barrier synchronization after executing the preamble. That is, each host must execute the preamble before it begins executing the application but it need not wait for all other nodes to do so before proceeding with application execution.

The DAC deployment process proceeds as follows. The application and preamble code is (cross-)compiled and distributed to the deployment hosts (and the LAMINAR runtime is started on each). Or, if necessary, because a network copy is not feasible, the preamble is executed on each host. Then the application body's execution is initiated by presenting program inputs to the body's input nodes. To change the configuration of and to redeploy a distributed application, the developer need only change the node-to-host mappings (via the add_node API) and recompile/distribute the preamble to the hosts in the deployment. Note that, at present, deployment or redeployment is a globally synchronized operation; we have not yet implemented a "rolling deployment" feature. The application must stop before the preamble outputs are redeployed and then restarted. We currently use scripts to automate this process as much as possible.

IV. EXPERIMENTAL METHODOLOGY

To illustrate the utility of deployment-as-code, we measure the energy usage of a distributed anomaly detection application for sensor telemetry data. The application is complex enough to demonstrate the value of deployment options and of using LAMINAR as a high-level dataflow representation for distributed multiscale IoT programming. However, this complexity makes it difficult or impossible to generate an equivalent implementation using an alternative technology (e.g. for comparative purposes). It might be possible to achieve similar overall functionality, but verifying that both implementations are the same is infeasible. Thus, our intention with this study is to demonstrate how our work enables exploring alternative deployment options via minimal code changes. That is, because our approach eases deployment, it allows the deployment engineer a greater latitude of choices from which, in this study, the most energy-efficient may be chosen.

The application consists of multiple anomaly detectors and an arbitrator (**ARB**) that decides whether to report an anomaly based on the detector outputs. We use three algorithms in this study. **KS** which compares the distribution of the most recent k values to the previous k values using a Kolmogorov-Smirnoff test and reports an anomaly when the distributions differ. **CORR** computes the linear correlation between the most recent k values and the previous set of k values and reports an anomaly when the correlation coefficient is not statistically significant. **REG** computes the linear regression coefficients and confidence bounds using the k most recent values as explained variables and the previous set of k values as explanatory values. It reports an anomaly when both coefficients are outside their respective confidence intervals.

The experiments stream data from a sensor to the detection algorithms on a fixed and periodic duty cycle. Each algorithm independently determines whether the most recent values constitute an abnormal condition and reports a binary value (i.e., anomaly or not-anomaly) to the arbitrator. The arbitrator implements a voting algorithm to determine whether it should report an anomaly. Figure 1 shows the LAMINAR (dataflow) representation of the application. Listing 1 shows the preamble (using our API) that constructs the executable representation, mapping nodes/operands onto two hosts (HOST1 and

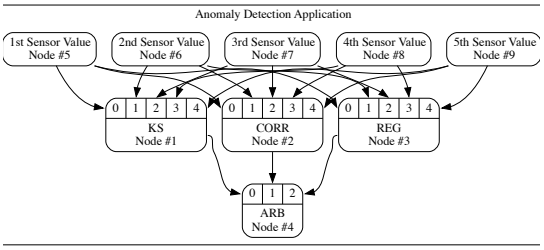

Fig. 1. Dataflow representation of the anomaly detection application.

HOST2). This representation places the operands and the KS node on HOST1; all other nodes are on HOST2. Reconfiguration requires only that the host assignments be updated.

Our experiments use a duty cycle for sensor telemetry of 12 seconds. Each anomaly detection algorithm computes the most recent $k = 5$ measurements. In this way, the application compares the most recent minute's measurements with the previous minute's measurements to determine if there has been a change in conditions.

Listing 1. LAMINAR Anomaly Application Preamble

```
laminar_init();
//specify the hosts in the deployment
//HOST1 and HOST2 are unique integers
add_host(HOST1, IP1, "/anomaly_app");
add_host(HOST2, IP2, "/anomaly_app");
// create the application dataflow nodes
// K_ID, R_ID, C_ID and A_ID are unique integers
add_node(HOST1,K_ID,{DF_CUSTOM,KS});
add_node(HOST2,R_ID, {DF_CUSTOM,REG});
add_node(HOST2,C_ID,{DF_CUSTOM,CORR});
add_node(HOST2,A_ID,{DF_CUSTOM,ARB});
// create the graph input nodes (operands)
// and link them to KS, REG, and CORR
int op_node = 5;
for(int IN=0; IN < 5; IN++) {
 // create input node
 add_operand(HOST1,op_node);
 // create edges to
 // KS, REG, and CORR
 subscribe(K_ID,IN,op_node);
 subscribe(R_ID,IN,op_node);
 subscribe(C_ID,IN,op_node);
 op_node++;
}
// create edges from KS, REG, and CORR to ARB
subscribe(A_ID,IN0,K_ID);
subscribe(A_ID,IN1,R_ID);
subscribe(A_ID,IN2,C_ID);
// construct the deployment for each host
laminar_setup(); //builds the logs
```

A. Experimental Setup

We measure the energy use by resource-constrained edge devices when executing different deployment configurations of the anomaly detection application. In all configurations, we place the **ARB** component on a cloud-hosted virtual machine (VM), which reports the presence or absence of an anomaly. The edge device generates sensor readings (simulated using a random floating point number generator) for the application.

We compare the energy requirements of LAMINAR using two different edge devices: an Arduino Feather Huzzah (to which we refer to as the Feather) and a Raspberry Pi 3B+. The Feather implements an 80Mhz ESP8266 microcontroller with 32KB/80KB instruction/data storage, and 4MB flash. The Pi 3B+ has a 1.4GHz Cortex-A53 processor, 1GB of memory, and 32GB flash. Both devices include a built-in 802.11 WiFi networking interface; however, the Feather is only able to operate at 2.4 GHz. Anecdotally, some routers could not correctly negotiate the Feather's frequency capability. For this reason, we conducted all experiments using a 2.4 GHz router. We used the same 5V/2.5A power supply in all experiments for both devices. We report results in terms of energy usage by the edge devices in different deployment configurations. All energy measurements were gathered using an Onset UX120-018 Hobo Plug Load Logger attached to a CanaKit 5V/2.5A power supply. The logger was set to record energy measurements every second (the finest resolution available).

The VM was an Ubuntu 20.04 image hosted in a campus cloud using the KVM hypervisor and a 1GB network interface. When instantiated, the instance was configured with 2 cores (2.5 GHz) and 8GB of memory. We used the same VM for all experiments. The **ARB** in all experiments was configured to use the default messaging and persistence modes (cf. Section II). When the device was configured to use MQTT, the MQTT broker (implemented by mosquitto version 1.6.9) was colocated with the **ARB**.

We connected the edge device via 802.11 to a local router, which connects to the common carrier Internet and the campus network, hosting the campus cloud and the VM. The LAMINAR MQTT transport is translated to ZeroMQ via the MQTT-LAMINAR runtime (MQTT-LRT) gateway colocated with the MQTT broker and the LAMINAR instance within the VM. Note that for all deployments tested, we chose to colocate the message gateway, the MQTT broker, and the ZeroMQ Laminar service but did not implement any further intra-machine message optimization.

In our experiments, we name each deployment configuration to indicate one of three options, separated by hyphens in each name. The first part of the name indicates which LAMINAR nodes were assigned to the edge device for that deployment. The letters "K," "R," and "C" indicate that **KS**, **REG**, and **CORR**, respectively, were hosted on the edge device. All other nodes are assigned to the VM. The word "none" indicates that all LAMINAR nodes were assigned to the VM, and the edge device only generates the input sensor data.

The second part of the name indicates whether the device used a delay or process sleep function (denoted "nosleep") or a deep-sleep function (denoted "sleep") to wait while idle until the beginning of a duty cycle. Finally, the third part of each name indicates whether the runtime was configured to use persistent storage (denoted "persist") or in-memory storage (denoted "memory"). For example, the name "K-nosleep-persist" refers to a configuration in which **KS** is assigned to the edge device, the device actively waits during idle periods between duty cycles using a delay or process sleep function, and the runtime uses persistent storage for its logs.

V. RESULTS

We next show how deployment-as-code can be used to capture the energy consumption required by various deployments of the anomaly detection application. We consider both edge devices with and without a firewall. Note that all the

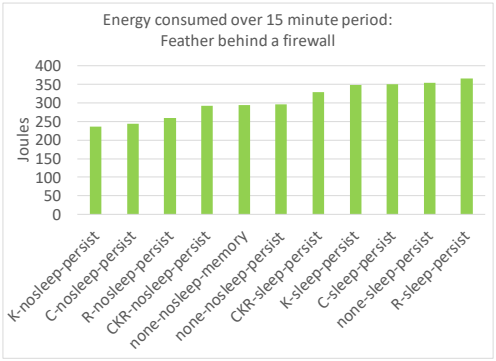

Fig. 2. Total energy consumed (in Joules) for different LAMINAR deployments of the anomaly detection application on the Feather.

deployments used two hosts, designated as host 1 and host 2 in the program preamble. Thus, each deployment across *all of the experiments that follow* required only that the host identifier be changed to indicate the node-to-host mapping for the program nodes. There are no other code differences between deployments.

Figure 2 shows a comparison, in Joules, of the energy consumed by the Feather over 15 minutes, for the different deployment configurations of the application. To generate the results shown in the figure, we sited the Feather in a local residence (i.e. a student apartment) serviced by residential Internet connectivity and isolated by a firewall. For each configuration (along the x-axis), the figure shows the consumed energy, in Joules, on the y-axis. In each case, we ran the application for 30 minutes but recorded the energy consumption during the last 15 minutes to allow for thermal stabilization of the device and the power supply.

Surprisingly, energy consumption is minimized when **KS** is assigned to the Feather rather than when all computational nodes (i.e., the "none" configurations) are assigned to the VM. Indeed, even when **CORR**, **KS**, and **REG** are all executed on the Feather, the energy consumption is lower than when none of them are. This result indicates that the LAMINAR implementation using an MQTT-LRT gateway expends more energy communicating inputs to **CORR**, **KS**, and **REG** than it does when executing them.

To understand why, consider the graphical dataflow representation of the application shown in Figure 1. **KS**, **CORR** and **REG** each require 5 input values to fire. Because there are no data dependencies between these nodes, they can execute in parallel even when the target platform is single-threaded and not timeshared (as is the case when executing on the Feather). Thus, before LAMINAR fires each node, it must gather and deliver the node's inputs. In the current prototype, nodes store their outputs in logs collocated with the node's execution. Thus in the "none" configurations, LAMINAR fetches the inputs for **KS**, **CORR**, and **REG** from the edge device to the VM where they will each be executed. Further, the VM implementation attempts to execute these nodes in parallel, so LAMINAR performs these fetches independently. The result is that the 5 sensor values are sent 15 times from the edge to the VM per duty cycle. Note that **ARB** takes only a single boolean value from **KS**, **CORR**, and **REG**. As a result, a configuration in which one of these anomaly test nodes is assigned to the edge reduces the energy consumed by the device required to transport 4 messages across the network and increases the energy consumption for node execution.

It is also noteworthy that using the deep-sleep functionality available for the Feather requires more energy than an active idle waiting implementation. This result is likely due to the Feather's design. Specifically, the deep sleep mode for this device de-energizes all on-board subsystems except a timer. The result is that the Feather goes through a complete power-up and restart when awakened from a deep sleep. Note that power consumption is approximately an order of magnitude less during deep sleep than during normal operation.

With the runtime configured for persistence, LAMINAR will resume and continue the application after power-up or restart so it supports this power-cycle deep sleep mode transparently. However, from an energy consumption perspective, because the network interface completely powers down during a deep sleep, the device must reacquire a wireless connection and then reestablish an MQTT connection after each deep sleep. Because the duty cycle is 12 seconds in these experiments, the deployments that enable deep sleep use more energy (due to the energy required to reacquire the network and MQTT connectivity) compared to active waiting.

Finally, but unsurprisingly, the deployment that uses an in-memory implementation of logs uses less energy than one that uses persistence. Note that the Feather can support a maximum of $80KB$ of program data in memory. As such, there is insufficient program memory to hold the logs used by LAMINAR for all but a "none" deployment. Further, LAMINAR cannot pause and resume application execution using in-memory logs when power is lost. Thus, using the in-memory configuration with the deep sleep configuration is impossible. Comparing "none-nosleep-memory" to "none-nosleep-persistence" shows that the additional energy needed to store and retrieve data from persistent storage on the Feather is not substantial over a 15 minute period.

Note that we report the total energy used rather than the energy used per unit time to provide insight into the impact on the battery charging lifecycle. Specifically, the energy used during battery operation (and not the power) correlates with discharge time. Because we use the same 5 volt power supply for all experiments (and the internal voltage used by the Feather is 3.3 volts) and because the plug logger cannot measure the voltage of the device internally, we also do not report amp-hours. In a battery-powered context using a Feather, the battery would connect the Feather's onboard 3.3 volt regulator and not the 5 volt mini USB connector, and we do not have access to the energy dissipation caused either by the step-down transformer in the USB interface nor the on-board voltage regulator circuit. However, because we report energy totals, the differences in energy consumed shown in Figure 2 are a function of experiment duration.

While predicting battery discharge duration is always challenging, particularly for outdoor settings where environmental

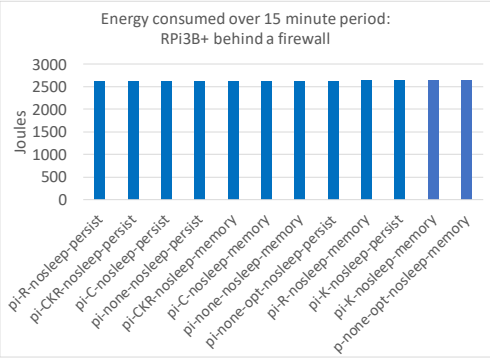

Fig. 3. Total energy consumed (in Joules) for the LAMINAR anomaly detection application on the RPi3B+ behind a firewall.

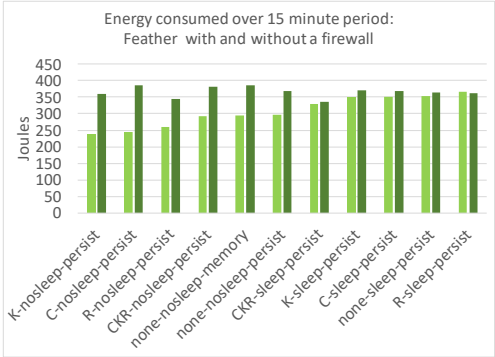

Fig. 4. Comparison of total energy consumed (in Joules) by the Feather when connected to two different networks, one behind a firewall (light green) and the other not (dark green).

conditions can affect discharge rates, it is possible to compare theoretically achievable discharge times from the data shown in Figure 2. Specifically, for a $2000mAh$ Lithium Poly battery, the average energy consumption of the "K-nosleep-persist" configuration (the most energy-efficient of those tested) corresponds to a maximum discharge time of approximately 25 hours compared to approximately 16.8 hours for "R-sleep-persist" (the least energy efficient).

These results illustrate two essential features enabled by deployment-as-code. The first is that the same application code paired with different preambles, each specifying a different deployment configuration as node-to-host assignments, results in a working deployment. Note that the execution of a preamble results in a set of logs that serve as input to the application body. Secondly, the choice of deployment for even a simple application can affect energy efficiency substantially, and deployment-as-code facilitates the exploration of these effects through simple node-to-host mappings in the preamble.

A. RPi3B+ as Edge Device

Figure 3 shows the energy consumption for the anomaly-detection application when executed on a Raspberry Pi 3B+ (RPi3B+). The device is sited in the same location (i.e., in a residence behind a firewall) as in the previous experiment.

For these results, we used the same deployment setting as for those depicted in Figure 2. In Figure 3, all deployments that used the RPi3B+ are prefixed with "pi-". Note that the RPi3B+ does not include a deep-sleep mode, and we did not have a straightforward way to implement a timed power-off period to emulate the deep-sleep mode of the Feather. The active delay on the RPi3B+ was implemented using the Linux `sleep` system call but the same as the Feather implementation otherwise. That is, the runtime was single-threaded, persistent storage did not use virtual memory, and communication between the edge device and the VM used the MQTT-LRT gateway and MQTT broker.

The results shown in Figure 3 indicate that because the RPiB3+ does not implement power or energy management, all LAMINAR application configurations require essentially the same amount of energy (the differences, on the scale shown, are virtually indistinguishable). However, the energy usage, by comparison, is substantially greater than that of the Feather.

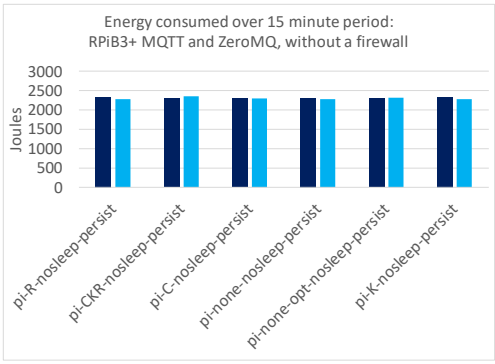

Fig. 5. Comparison of total energy consumed (in Joules) by RPiB3+ using two different networks, behind a firewall (blue) and not (dark blue).

This observation illustrates how results gathered from one device architecture do not necessarily translate to another as a function of deployment configuration. For example, simple intuition would indicate that a Feather would require less energy than an RPi3B+. What is less obvious is that the amount of savings is a function of deployment configuration for the Feather alone.

B. Effect of Unseen Configurations

Figure 4 compares the energy consumption for the Feather for LAMINAR configurations behind a firewall (same data as in Figure 2) to those that employ a different network (without a firewall, shown in dark green). For the latter experiments, we connected a a TP-Link Omada EAP-110-Outdoor wireless access point directly to the campus network. We chose this access point since it is one we have used extensively in several outdoor field deployments. The access point was configured not to provide DHCP or NAT service, so the Feather received a campus network address from the DHCP server for the subnetwork to which the access point was connected. Note that this subnetwork differed from the one hosting the VM (i.e., all network traffic traversed the campus intranet). The Feather was the only client attached to the access point during the experiments. The only difference between the light and green bars is the wireless network and device location. The deployment configurations are shown on x-axis.

From the figure, somewhat surprisingly, the energy usage by the Feather is higher in the campus setting where the access point and the network were significantly lower latency than in the home setting with a firewall. Moreover, deep sleep mode appears to use less energy compared to active waiting, whereas on the home setting, it used more energy (although the difference is small). This latter observation is especially puzzling since reestablishing network connectivity after a deep sleep requires a response from a campus-level DHCP server (and not just the local access point, as in the home setting). In short, access to a lower-latency network, which should have reduced communication time (and saved energy) required more energy compared to a slower, less performant network.

We believe that this somewhat counterintuitive result is due to an 802.11 power-saving configuration that is optionally supported by different access points. Specifically, the ESP8266 microcontroller can use a modem sleep mode between 802.11 DTIM beacon intervals, which were longer in the home setting than in the campus setting.

Modem sleep between DTIM intervals is documented, but the configuration of the access points with which the Feather was communicating in these experiments was not easily discoverable. Deployment-as-code, however, allowed us to discover the most energy-efficient deployments through simple mapping changes in each preamble.

In Figure 5, we show the effect of 802.11 connectivity on energy consumption for the RPiB3+. The figure compares the results for the RPiB3+ (from Figure 3, shown in blue) for the home setting to those for the campus setting (dark blue) using the TP-Link access point. Note that we observed no difference between persistent and in-memory deployment configurations in either setting, so we show only the results for the persistent configurations. The only difference in the experimental setup between the use of the two devices was the location, and network connectivity.

These results confirm the conventional intuition that a lower-latency network improves energy usage for the RPi3B+. Otherwise, the results confirm that for the RPi3B+, the deployment configuration of this application does not affect energy consumption significantly. Deployment-as-code enables us to easily compare a wide range of deployment options with only minimal changes to the DAC preamble (node to host assignments) in a network transparent way.

VI. Conclusions

In this work, we present a new approach to distributed deployment of IoT application components (called "deployment-as-code") that leverages the dataflow programming model. Our extensions include support for microcontrollers, lightweight log implementations, and decoupling (separate compilation) of the application preamble from its body. Our evaluation shows that this approach simplifies deployment experimentation and facilitates greater understanding of energy efficiency and performance of edge/fog applications.

References

[1] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, p. 44–54, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3368454

[2] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[3] M. Chadha, V. Pacyna, A. Jindal, J. Gu, and M. Gerndt, "Migrating from microservices to serverless: an iot platform case study," in *International Workshop on Serverless Computing*, 2022, p. 19–24.

[4] T. Ekaireb, L. Brand, N. Avaraddy, M. Mock, C. Krintz, and R. Wolski, "Distributed dataflow across the edge-cloud continuum," in *IEEE Conference on Cloud Computing*, 2024.

[5] B. Marr, J. Karl, L. Lewins, K. Prager, and D. Thompson, "An Asynchronous Dataflow Signal Processing Architecture to Minimize Energy per Op," in *ASYNC Symposium*, 2013.

[6] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. Von Platen, M. Mattavelli, and M. Raulet, "Opendf: a dataflow toolset for reconfigurable hardware and multicore systems," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 5, pp. 29–35, 2009.

[7] E. C. Klikpo, J. Khatib, and A. Munier-Kordon, "Modeling multi-periodic simulink systems by synchronous dataflow graphs," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016, pp. 1–10.

[8] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *EuroSys*, 2007.

[10] D. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "CIEL: a universal execution engine for distributed data-flow computing," in *USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[11] "Mqtt - the standard for iot messaging," 2024, https://mqtt.org/.

[12] Arvind, K. P. Gostelow, and W. Plouffe, "Indeterminacy, monitors, and dataflow," in *Symposium on Operating System Principles*, 1977.

[13] J. R. Gurd, "The manchester dataflow machine," *Computer Physics Communications*, vol. 37, no. 1-3, pp. 49–62, 1985.

[14] D. E. Culler and Arvind, "Resource requirements of dataflow programs," *ACM SIGARCH Computer Architecture News*, vol. 16, no. 2, pp. 141–150, 1988.

[15] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A report on the sisal language project," *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, pp. 349–366, 1990.

[16] Y. Bu, B. Howe, M. Balazinska, and M. Ernst, "HaLoop: Efficient iterative data processing on large clusters," in *Proc. VLDB Endow.*, 2010.

[17] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientfic analysis," in *eScience*, 2008.

[18] J. Backus, "Programming language semantics and closed applicative languages," in *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on principles of programming languages*, 1973, pp. 71–86.

[19] R. Wolski, C. Krintz, F. Bakir, G. George, and W.-T. Lin, "CSPOT: Portable, Multi-scale Functions-as-a-Service for IoT," in *ACM Symposium on Edge Computing*, 2019.

[20] W.-T. Lin, C. Krintz, and R. Wolski, "Tracing Function Dependencies Across Clouds," in *IEEE Cloud*, 2018.

[21] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis, "Corfu: A shared log design for flash clusters," in *USENIX NSDI*, 2012.

[22] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, "Chariots: A scalable shared log for data management in multi-datacenter cloud environments." in *EDBT*, 2015, pp. 13–24.

[23] E. Ozdogan, O. Ceran, and M. Ustunag, "Systematic analysis of infrastructure as code technologies," *GU J. Sci.*, vol. 10, no. 4, 2023.

[24] F. Bakir, C. Krintz, and R. Wolski, "Caplets: Resource aware, capability-based access control for iot," in *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2021, pp. 106–120.

[25] "Littlefs," 2023, https://github.com/littlefs-project/littlefs.

[26] Sun Microsystems, "Write-Once-Run-Anywhere (WORA)," 2024, "https://en.wikipedia.org/wiki/Write_once,_run_anywhere".