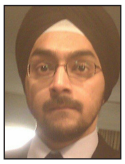
# It's Time to End Monolithic Apps for Connected Devices

RAYMAN PREET SINGH, CHENGUANG SHEN, AMAR PHANISHAYEE, AMAN KANSAL, AND RATUL MAHAJAN



Rayman Preet Singh is a PhD candidate in computer science at the University of Waterloo, Canada. He is co-advised by S. Keshav and Tim Brecht, and has broad research interests in distributed systems and ubiquitous computing. rmmathar@uwaterloo.ca

Chenguang Shen is a PhD candidate in computer science at the University of California, Los Angeles (UCLA), working with Professor Mani Srivastava. He obtained his MS in computer science from UCLA in 2014, and a BEng in software engineering from Fudan University, Shanghai, China in 2012. Chenguang's research focuses on developing a mobile sensing framework for context awareness. cgshen@cs.ucla.edu

Amar Phanishayee is a Researcher at Microsoft Research. His research efforts center around rethinking the design of datacenter-based systems, from infrastructure for compute, storage, and networking to distributed systems that are scalable, robust to failures, and resource-efficient. He is also interested in storage and programming abstractions for connected devices. Amar received his PhD from Carnegie Mellon University in 2012. amar@microsoft.com

The proliferation of connected sensing devices (or Internet of Things) can in theory enable a range of "smart" applications that make rich inferences about users and their environment. But in practice, developing such applications today is arduous because they are constructed as monolithic silos, tightly coupled to sensing devices, and must implement all data sensing and inference logic, even as devices move or are temporarily disconnected. We present Beam, a framework and runtime for distributed inference-driven applications that breaks down application silos by decoupling their inference logic from other functionality. It simplifies applications by letting them specify "what should be sensed or inferred," without worrying about "how it is sensed or inferred." We discuss the challenges and opportunities in building such an inference framework.

Connected sensing devices such as cameras, thermostats, and in-home motion, door-window, energy, and water sensors, collectively dubbed the Internet of Things (IoT), are rapidly permeating our living environments, with an estimated 50 billion such devices projected for use by 2020 [2]. They enable a wide variety of applications spanning security, efficiency, healthcare, and others. Typically, these applications collect data using sensing devices to draw inferences about the environment or the user, and use these inferences to perform certain actions. For example, Nest uses motion sensor data to infer and predict home occupancy and adjusts the thermostat accordingly.

Most IoT applications today are being built in a monolithic way. That is, applications are tightly coupled to the hardware. For instance, Nest's occupancy prediction can only be used with the Nest device. Applications need to implement all the data collection, inferencing, and user functionality-related logic. For application developers, this increases the complexity of development, and hinders broad distribution of their applications because the cost of deploying their specific hardware limits user adoption. For end users, each sensing device they install is limited to a small set of applications, even though the hardware capabilities may be useful for a broader set of applications. How do we break free from this monolithic and restrictive setting? Can we enable applications to be programmed to work seamlessly in heterogeneous environments with different types of connected sensors and devices, while leveraging devices that may only be available opportunistically, such as smartphones and tablets?

To address this problem, we start from the insight that many inferences required by applications can be drawn using multiple types of connected devices. For instance, home occupancy can be inferred using motion sensors (e.g., those in security systems or in Nest), cameras (e.g., Dropcam), microphone, smartphone GPS, or using a combination of these, since each may have different sources of errors. Therefore, *we posit that inference logic, traditionally left up to applications, ought to be abstracted out as a system service*. Such a service will relieve application developers of the burden of implementing and training commonly used

Aman Kansal received his PhD in electrical engineering from the University of California Los Angeles, where he was honored with the department's Outstanding PhD Award. His current research interests include all aspects of sensing systems, with a special emphasis on embedded sensing, context inference, and energy efficiency. He has published over 65 papers, and his work has also been recognized with the Microsoft Gold Star award. kansal@microsoft.com

Ratul Mahajan is a Principal Researcher at Microsoft Research and an Affiliate Professor at the University of Washington. His research interests include all aspects of networked systems. His current work focuses on software-defined networks and network verification, and his past work spans Internet routing and measurements, incentive-compatible protocol design, practical models for wireless networks, vehicular networks, and connected homes. ratul@microsoft.com

inferences. More importantly, it will enable applications to work using any of the sensing devices that the shared inference logic supports.

We surveyed and analyzed two popular application *classes* in detail, one that infers environmental attributes and another that senses an individual user.

- *Rules*: A large class of applications is based on the *If This Then That* (IFTTT) pattern [1, 8]. IFTTT enables users to create their own rules that map sensed attributes to desired actions. We consider a particular rules application that alerts a user if a high-power appliance, e.g., electric oven, is left on when the home is unoccupied. This application uses the appliance-state and home occupancy inferences.
- *Quantified Self* (QS) captures a popular class of applications that disaggregate a user's daily routine by tracking her physical activity (walking, running, etc.), social interactions (loneliness), mood (bored, focused), computer use, and more.

In analyzing these two popular classes of applications, we identify the following three key challenges for the proposed inference service:

**1. Decouple applications, inference algorithms, and devices**: Data-driven inferences can often be derived using data from multiple devices. Combining inputs from multiple devices, when available, generally leads to improved inference accuracy (often overlooked by developers). Figure 1 illustrates the improvement in inference accuracy for the occupancy and physical activity inferences, used in the Rules and Quantified Self applications, respectively; 100% accuracy maps to manually logged ground truth over 28 hours.

Hence, applications should not be restricted to using a single sensor or a single inference algorithm. At the same time, applications should not be required to incorporate device discovery, handle the challenges of potentially using devices over the wide area (e.g., remote I/O and tolerating disconnections), use disparate device APIs, and instantiate and combine multiple inferences depending on available devices. Therefore, an inference framework must decouple (1) "what is sensed" from "how it is sensed" and (2) "what is inferred" from "how it is inferred." It should require an application to only specify the desired inference, e.g., occupancy (in addition to inference parameters like sampling rate and coverage), while handling the complexity of configuring the right devices and inference algorithms.

**2. Handle environmental dynamics**: Applications are often interested in tracking user and device mobility, and adapting their processing accordingly. For instance, the QS application needs to track which locations a user frequents (e.g., home, office, car, gym, meeting room, etc.), handle intermittent connectivity, and more. Application development stands to be greatly simplified if the framework were to seamlessly handle such environmental dynamics, e.g., automatically update the selection of devices used for drawing inferences based on user location. Hence the QS application, potentially running on a cloud server, could simply subscribe to the activity inference, which would be dynamically composed of sub-inferences from various devices tracking a user.

**3. Optimize resource usage**: Applications often involve continuous sensing and inferring, and hence consume varying amounts of system resources across multiple devices over time. Such an application must structure its sensing and inference processing across multiple devices, in keeping with the devices' resource constraints. This adds undue burden on developers. For instance, in the QS application, wide area bandwidth constraints may prevent backhauling of high rate sensor data. Moreover, whenever possible, inferences should be shared across multiple applications to prevent redundant resource consumption. Therefore, an inference framework must not only facilitate sharing of inferences, but in doing so must optimize for efficient resource use (e.g., network, battery, CPU, memory, etc.) while meeting resource constraints.

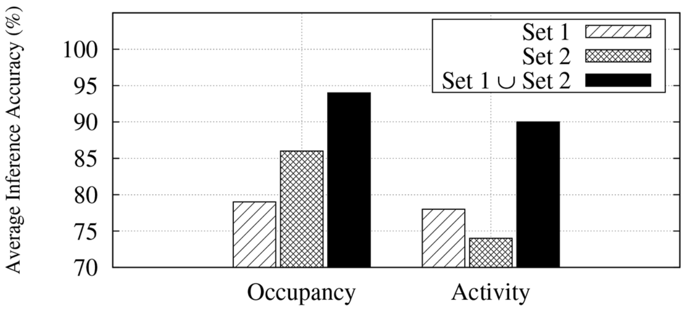


**Figure 1**: Improvement in occupancy and activity inference accuracy by combining multiple devices. For occupancy, sensor set 1 = {camera, microphone} in one room and set 2 = {PC interactivity detection} in a second room. For physical activity, set 1 = {phone accelerometer} and set 2 = {wrist worn FitBit}.

## Beam Inference Framework

To explore the above challenges concretely, we propose Beam, an application framework and associated runtime for data-driven inference-based applications. Beam provides applications with inference-based programming abstractions. Applications subscribe to high-level inferences, and Beam dynamically identifies the required sensors in the given deployment and constructs an appropriate *inference graph*. The inference graph is made up of *modules,* which are processing units that encapsulate inference algorithms; modules can use the output of other modules for their processing logic. The Beam runtime instantiates the inference graph to initiate data processing on suitable devices. Beam's user-tracking service and optimizer mutate this graph at runtime for handling environment dynamics and for efficient resource usage, respectively.

Beam introduces three simple abstractions that are key to constructing and maintaining the inference graph. First, typed *inference data units* (IDUs) guide module composability. Modules can be linked to accept IDUs from other modules and generate IDUs. Second, *channels* abstract all inter-module interaction, allowing Beam to seamlessly migrate modules and mask transient disconnections when interacting modules are not collocated. Third, *coverage tags* provide a flexible and low-overhead way to connect sensors with the right coverage characteristics (e.g., location, users) to applications. We describe these key abstractions in detail next.

**Inference graphs**: Inference graphs are directed acyclic graphs that connect sensors to applications. The nodes in this graph correspond to *inference modules* and edges correspond to *channels* that facilitate the transmission of IDUs between modules. Figure 2 shows an example inference graph for the Quantified Self application that uses eight different devices spread across the user's home and office and includes mobile and wearable devices.

Composing an inference as a directed graph enables sharing of data-processing modules across applications and across modules that require the same input. In Beam, each compute device associated with a user, such as a tablet, phone, PC, or home hub, has a part of the runtime, called the *engine*. Engines host inference graphs and interface with other engines. Figure 3 shows two engines, one on the user's home hub and another on his phone; the inference graph for QS shown earlier is split across these engines, with the QS application itself running on a cloud server. For simplicity, we do not show other engines such as one running on the user's work PC.

**IDU**: An *inference data unit* (IDU) is a typed inference, and in its general form is a tuple <t,s,e>, which denotes any inference with state information *s*, generated by an inference algorithm at time *t* and error *e*. The types of the inference state *s* and error *e*, are specific to the inference at hand. An example IDU is (09/23/2015 10:10:00, occupied, 90%). Inference state *s* may be of a numerical

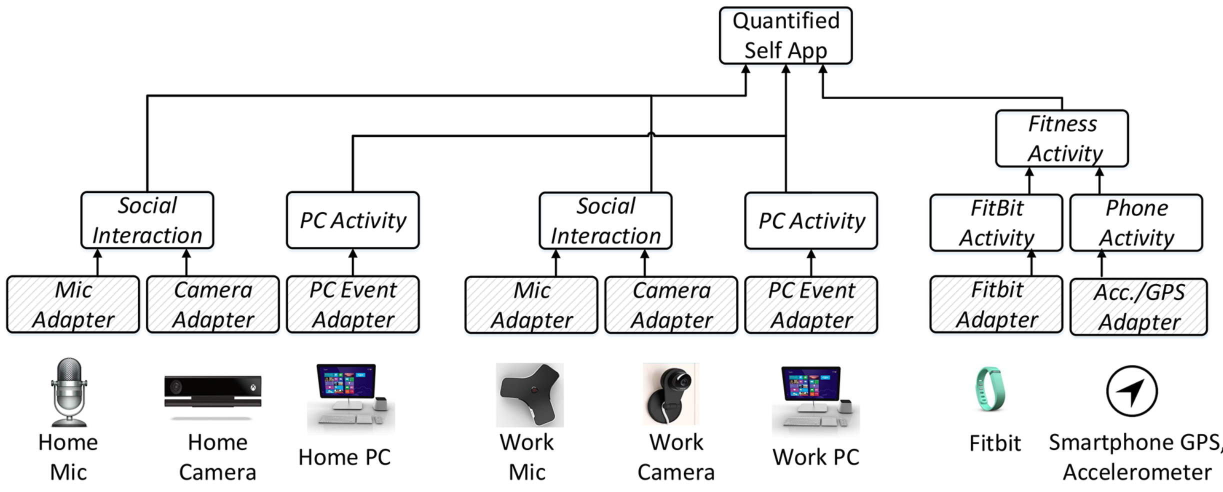


**Figure 2:** Inference graph for Quantified Self app

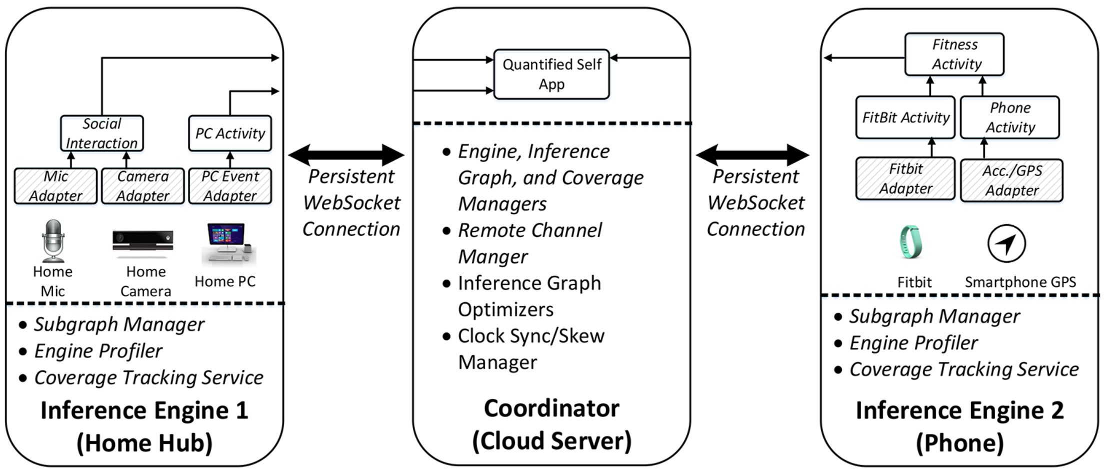


**Figure 3:** An overview of different components in an example Beam deployment with two engines

type such as a double (e.g., inferred energy consumption); an enumerated type such as high, medium, low; or numerical types. Similarly, error *e* may specify a confidence measure (e.g., standard deviation), probability distribution, or error margin (e.g., radius). IDUs abstract away "what is inferred" from "how it is inferred." The latter is handled by inference modules, described next.

**Inference modules**: Beam encapsulates inference algorithms into typed modules. Inference modules consume IDUs from one or more modules, perform certain computations using IDU data and pertinent in-memory state, and output IDUs. Special modules called *adapters* interface with underlying sensors and output sensor data as IDUs. Adapters decouple "what is sensed" from "how it is sensed." Module developers specify the IDU types a module consumes, the IDU type it generates, and the module's input dependency (e.g., {PIR} OR {camera AND mic}). Modules have complete autonomy over how and when to output an IDU and can maintain arbitrary internal state. For instance, an occupancy inference module may (1) specify input IDUs from microphone, camera, and motion sensor adapters, (2) allow multiple microphones as input, and (3) maintain internal state to model ambient noise.

**Channels**: To ease inference composition, *channels* link modules to each other and to applications. They encapsulate the complexities of connecting modules across different devices, including dealing with device disconnections and allowing for optimizations such as batching IDU transfers for efficiency. Every channel has a single *writer* and a single *reader* module. Modules can have multiple input and output channels. Channels connecting modules on the same engine are *local*. Channels connecting modules on two different engines, across a local or wide area network, are *remote* channels. They enable applications and inference modules to seamlessly use remote devices or remote inference modules.

**Coverage tags**: Coverage tags help manage sensor coverage. Each adapter is associated with a set of coverage tags that describe what the sensor is sensing. For example, a location string tag can indicate a coverage area such as "home," and a remote monitoring application can use this tag to request an occupancy inference for this coverage area. Coverage tags are strongly typed. Beam uses tag types only to differentiate tags and does not dictate tag semantics. This allows applications complete flexibility in defining new tag types. Tags are assigned to adapters at setup time using inputs from the user, and are updated at runtime to handle dynamics.

Beam's runtime also consists of a *coordinator*, which interfaces with all engines in a deployment and runs on a server that is reachable from all engines. The coordinator maintains remote channel buffers to support reader or writer disconnections (typical for mobile devices). It also provides a place to reliably store state of inference graphs at runtime while being resistant to engine crashes and disconnections. The coordinator is also used to maintain reference time across all engines. Engines interface with the coordinator using a persistent Web-socket connection, and instantiate and manage local parts of an inference graph(s).

## *Beam Runtime*

Beam creates or updates inference graphs when applications request inferences, mutates the inference graphs appropriately to handle environmental dynamics, and optimizes resource usage.

**Inference graph creation**: An application may run on any user device, and the sensors required for a requested inference may be spread across devices. Applications request their local Beam

| Function Application | Description |
|---|---|
| **APIs:** | |
| Request(InferenceModule, List<Tag>, Mode, [SamplingRate]) Request(InferenceType, List<Tag>, Mode, [SamplingRate]) | Returns a channel to specified module or to a module that outputs specified inference (and instantiates the inference graph) |
| CancelRequest(InferenceModule) | Delete channel to specified module, and terminate its inference graph |
| **Channel APIs:** | |
| DeliverCallback(Channel, List<IDU>) | Receive a list of IDUs (invoked *on* channel reader) |
| Start(), Stop() | Start or stop a channel (invoked by channel reader) |
| **Inference Module APIs:** | |
| Initialize(ModuleSpec, [SamplingRate]) | Initialize the module with given specification and reporting rate |
| PushToOutputChannels(IDU) | Push inference data unit (IDU) to all output channels |
| AllOutputChannelsStopped() | Stop sensing/processing because all output channels stopped |
| OutputChannelRestarted(Channel) | Restart sensing/processing because an output channel is restarted |
| **Optimizer APIs:** | |
| UpdateGraphs(List<Graph>, List<Engine>, App, Req/Cancel, Module, [Mode]) | Incorporates module request and returns updated list of inference graphs |
| ReevaluateGraphs(List<Graph>, List<Engine>) | Returns updated list of inference graphs (new incarnation) after reevaluation |

**Table 1:** Key Beam APIs: Beam offers APIs for application, inference module, and optimizer developers. Applications and inference modules use channels for communication. [] denotes an optional parameter.

engine for all inferences they require. All application requests are forwarded to the coordinator, which uses the requested inference to look up the required module. It recursively resolves all required inputs of that module (as per its specification) and reuses matching modules that are already running. The coordinator maintains a *set* of such inference graphs as an *incarnation*. The coordinator determines where each module in the inference graph should run and formulates the new incarnation. The coordinator initializes buffers for remote channels, and partitions the inference graphs into engine-specific subgraphs, which are sent to the engines.

Engines receive their respective subgraphs, compare each received subgraph to existing ones, and update them by terminating deleted channels and modules, initializing new ones, and changing channel delivery modes and module sampling rates as needed. Engines ensure that exactly one inference module of each type with a given coverage tag is created.

**Inference delivery and guarantees**: For each inference request, Beam returns a channel to the application. The inference request consists of (1) required inference type or module, (2) *delivery mode*, (3) coverage tags, and (4) sampling requirements (optional).

Delivery mode is a channel property that captures data transport optimizations. For instance, in the *fresh push* mode, an IDU is delivered as soon as the writer-module generates it, while in the *lazy push* mode, the reader chooses to receive IDUs in batches, thus amortizing network transfer costs from battery-limited devices. Remote channels provide IDU delivery in the face of device disconnections by using buffers at the coordinator and the writer engine. Channel readers are guaranteed (1) no duplicate IDU delivery and (2) FIFO delivery based on IDU timestamps. Currently, remote channels use the *drop-tail* policy to minimize wide-area data transfers in the event of a disconnected/lazy reader. This means that when a reader reconnects after a long disconnection, it first receives old inference values followed by more recent ones. A *drop-head* policy may be adopted to circumvent this, at the cost of increased data transfers.

When requesting inferences, applications use tags to specify coverage requirements. Furthermore, an application may specify sampling requirements as a latency value that it can tolerate in detecting the change of state for an inference (e.g., *walking* periods of more than one minute). This allows adapters and modules to temporarily halt sensing and data processing to reduce battery, network, CPU, or other resources.

Channels and modules do not persist data. Applications and modules may use a temporal datastore, such as Bolt [5], to make inferences durable.

**Optimizing resource use**: The Beam coordinator uses inference graphs as the basis for optimizing resource usage. The coordinator reconfigures inference graphs by remapping the engine on which each inference module runs. Optimizations are either performed *reactively* (i.e., when an application issues/cancels an inference request) or *proactively* at periodic intervals. Beam's default reactive optimization minimizes the number of remote channels, and proactive optimization minimizes the amount of data transferred over remote channels. Other potential optimizations can minimize battery, CPU, and/or memory consumption at engines.

When handling an inference request, the coordinator first incorporates the requested inference graph into the incarnation, reusing already running modules, and merging inference graphs if needed. For new modules, the coordinator decides on which engines they should run (by minimizing the number of remote channels).

Engines profile their subgraphs and report profiling data (e.g., per-channel data rate) to the coordinator periodically. The coordinator annotates the incarnation using this data and periodically reevaluates the mapping of inference modules to engines. Beam's default proactive optimization minimizes wide area data transfers.

**Handling dynamics**: Movement of users and devices can change the set of sensors that satisfy application requirements. For instance, consider an application that requires camera input from the device currently facing the user at any time, such as the camera on her home PC, office PC, smartphone, etc. In such scenarios, the inference graph needs to be updated dynamically. Beam updates the coverage tags to handle such dynamics. Certain tags such as those of *location* type (e.g., "home") can be assumed to be static (edited only by the user), while for certain other types, e.g., *user*, the sensed subject is mobile and hence the sensors that cover it may change.

The coordinator's *tracking service* manages the coverage tags associated with adapters on various engines. The engine's tracking service updates the user coverage tags as the user moves. For example, when the user leaves her office and arrives home, the tracking service removes the *user* tag from device adapters in the office, and adds them to adapters of devices deployed in the home. The tracking service relies on device interactions to track users. When a user interacts with a device, the tracking service appends the user's tag to the tags of all adapters on the device.

When coverage tags change (e.g., due to user movement and change in sensor coverage), the coordinator recomputes the inference graphs and sends updated subgraphs to the affected engines.

## Current Prototype

Our Beam prototype is implemented as a cross-platform portable service that supports .NET v4.5, Windows Store 8.1, and Windows Phone 8.1 applications. Module binaries are currently wrapped within the service, but may also be downloaded from the coordinator on demand.

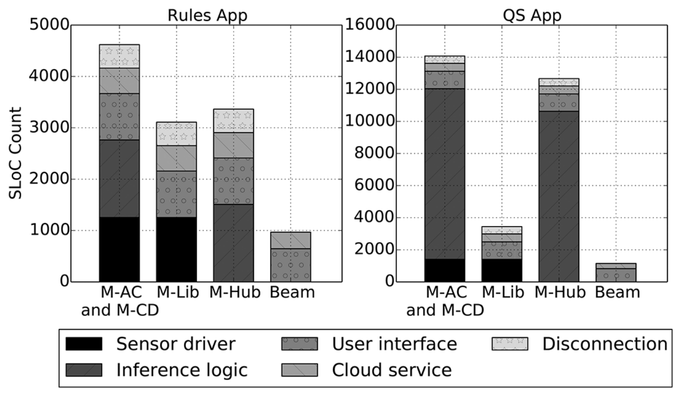


**Figure 4:** SLoC for different application components in the various development approaches

**APIs**: Table 1 shows the APIs that Beam exposes to application, inference module, and optimizer developers. Applications use the inference API to issue and cancel requests. Both inference modules and applications use the channel APIs to receive IDUs, and may *Stop* a channel to cease receiving IDUs. Each inference module is first initialized and provided with its specification and a sampling rate. It then begins its processing and pushes IDUs to all its output channels. If every output channel of a module is stopped, Beam informs the module (via AllOutputChannelsStopped), allowing it to stop its sensing/processing, thus saving resources until an output channel is restarted. Moreover, Beam abstracts optimization logic out of the coordinator, which allows modular replacement of proactive and reactive optimizers. Table 1 shows the inference graph management APIs that optimizers should implement to interface with Beam.

**Inferences**: We have implemented eight inference modules (mic-occupancy, camera-occupancy, appliance-use [3], occupancy, PC activity [6], fitness activity [7], semantic location, and social-interaction) and nine adapters (tablet and PC mic, powermeter, FitBit, GPS, accelerometer, PC interaction, PC event, and a HomeOS [4] adapter) to access all its device drivers.

**Sample applications**: We have implemented the two sample applications, *Rules* and *QS*, discussed earlier. Applications run on a cloud VM; Beam hosts the respective inference modules across the user's home PC, work PC, and phone.

Figure 4 compares the source lines of application code (SLoC) used in building these applications when using Beam and other development approaches. A monolithic approach where all sensor data is backhauled to a cloud-hosted application is denoted by *M-AC*. *M-CD* denotes an approach where a developer divides inference processing into fixed components that run on a cloud VM and end devices. *M-Lib* is similar to M-CD, except that an inference algorithm library is used. *M-Hub* denotes application development using device abstractions provided by the OS, e.g., HomeOS [4]. Moreover, we categorize the measured SLoC into the following different categories: (1) sensor drivers (one per sensor type); (2) inference algorithms, feature extraction, and learning models; (3) any required cloud-hosted services (as per the development approach) such as a storage, authentication, or access-control service; (4) mechanisms to handle device disconnections; and (5) user interface components, e.g., for displaying results or configuring devices. Using Beam results in up to 12x lower SLoC. Moreover, Beam’s handling of environmental dynamics results in up to 3x higher inference accuracy, and its dynamic optimizations match hand-optimized versions for network resource usage.

## Future Directions

Our experience in building the current Beam prototype has raised interesting questions and helped us identify various directions for future work.

Beam’s current tracking service only supports tracking of users (through device interactions) and mobile devices. We aim to extend tracking support to generic objects using passive tags such as RFID or QR codes.

Similarly, we aim to enrich Beam’s optimizers to include optimizations for battery, CPU, and memory. The key challenge in doing so lies in dynamically identifying the appropriate optimization objective (e.g., network, battery), issuing reconfigurations of inference graphs, while preventing hysteresis in the system.

Many in-home devices possess actuation capabilities, such as locks, switches, cameras, and thermostats. Applications and inference modules in Beam may want to use such devices. If the inference graph for these applications is geo-distributed, timely propagation and delivery of such actuation commands to the devices becomes important and raises interesting questions of what is the safe thing to do if an actuation arrives “late.”

Lastly, by virtue of its inference-driven interface, Beam enables better information control. A user can, in theory, directly control the inferences a given application can access. In contrast, existing device abstractions only allow the user to control the flow of device data to applications, with little understanding of what information is being handed over to applications. We hope to investigate the implications of this new capability in future work.

## Conclusion

Applications today are developed as monolithic silos, tightly coupled to sensing devices, and need to implement extensive data sensing and inference logic, even as devices move or have intermittent connectivity. Beam presents applications with inference-based abstractions and (1) decouples applications, inference algorithms, and devices; (2) handles environmental dynamics; and (3) optimizes resource use for data processing across devices. This approach simplifies application development, and also maximizes the utility of user-owned devices, thus surpassing current monolithic siloed approaches to building apps that use connected devices.

### *References*

[1] IFTTT: Put the Internet to work for you: https: //ifttt.com/.

[2] The Internet of Things: http://share.cisco.com/internet-of-things.html/.

[3] N. Batra, J. Kelly, O. Parson, H. Dutta, W. J. Knottenbelt, A. Rogers, A. Singh, and M. Srivastava, “NILMTK: An Open Source Toolkit for Non-Intrusive Load Monitoring,” in *Proceedings of the 5th International Conference on Future Energy Systems*, 2014.

[4] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl, “An Operating System for the Home,” in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’12)*, 2012.

[5] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan, “Bolt: Data Management for Connected Homes,” in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’14)*, April 2014.

[6] G. Mark, S. T. Iqbal, M. Czerwinski, and P. Johns, “Bored Mondays and Focused Afternoons: The Rhythm of Attention and Online Activity in the Workplace,” in *Proceedings of the ACM CHI*, 2014.

[7] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava, “Using Mobile Phones to Determine Transportation Modes,” *ACM Transactions on Sensor Networks*, vol. 6, no. 2, February 2010.

[8] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman, “Practical Trigger-Action Programming in the Smart Home,” in *Proceedings of the ACM CHI*, 2014.