A Framework for Internet of Things Search Engines Engineering

Nguyen Khoi Tran, M. Ali Babar
The University of Adelaide
Adelaide, SA 5005, Australia
nguyen.tran@adelaide.edu.au

Quan Z. Sheng
Macquarie University
Sydney, NSW 2109, Australia
michael.sheng@mq.edu.au

John Grundy
Monash University
Clayton, VIC 3800, Australia
john.grundy@monash.edu

***Abstract*—The content of the Internet of Things (IoT), notably sensor data and virtual representation of physical devices, has been increasingly delivered via Web protocols and available on the World Wide Web (WWW). Internet of Things Search Engine (IoTSE) systems are catalytic to utilize this influx of data. They enable users to discover and retrieve relevant IoT content. While a general IoTSE system – the next "Google" – is beyond the horizon due to the vast diversity of IoT content and types of queries for them, specific IoTSE systems that target subsets of query types and IoT infrastructure are feasible and beneficial. A component-based engineering approach, in which prior IoTSE systems and research prototypes are reassembled as building blocks for new IoTSE systems, could be a time- and cost-effective solution to engineering IoTSE systems. This paper presents the design, implementation, and evaluation of a framework to facilitate a component-based approach to engineering IoTSE systems. As an evaluation, we developed eight IoTSE components and composed them into eight proof-of-concept IoTSE systems, using a reference implementation of the proposed framework. An analysis on Source Line of Code (SLOC) revealed that the complexity handled transparently by the IoTSE framework could account for over 90% of the code base of a simple IoTSE system.**

***Index Terms*—Internet of Things, Search Engine, Framework, Architecture-centric, Microservice Architecture**

I. Introduction

By 2020, investment in Internet of Things (IoT) infrastructures could reach up to $832 billion[1]. These IoT infrastructures generate different types of content, such as sensory data, actuating services, and digital representations of physical entities, which could improve the efficiency of businesses and quality of life.

Internet of Things Search Engine (IoTSE) systems would be catalytic to realize the benefits of IoT infrastructure, as they enable humans and machines to retrieve the relevant IoT content, such as sensory data, actuating services and digital representatives of physical entities [1]. For instance, let's assume that a city council has deployed multi-purpose sensors across the city, which can measure temperature, light value, air quality, and sound level, among other data. Citizens could use this infrastructure to answer the queries for places and things in the city based on their real-time state via an IoTSE system. A typical query could be "Find a jogging trail near the city center, which would be fresh and quiet in the next hour." To answer this query, an IoTSE system would utilize historical and real-time sensor data to find sensors whose value would likely indicate "fresh and quiet" in the hour following the query. It would also use geological information stored in digital representations of jogging trails to find the ones near the city center. Finally, it would join two sets of results based on the spatial relations between sensors and running trails and respond to a user.

While existing Web search engines, sensor databases, and IoT cloud platforms have already offered some search capabilities, none of them could address all the specificities of IoT content comprehensively as a dedicated IoTSE system. The exemplary use case has demonstrated that IoTSE systems require complex inference and prediction capabilities, which are beyond the responsibility of IoT platforms and their rudimental look-up-by-ID ability. It has also highlighted that an IoT query can span multiple content collections and even cross organizational boundaries, both of which capabilities are beyond any individual sensor database. Moreover, the collections with which IoTSE systems work are generally not as well-controlled as document collections within databases. Therefore, IoTSE might also be required to crawl for new devices and content, update old content, and rebuild indexes regularly, similarly to Web search engines. However, IoTSE systems are different from Web search engines in two ways. First, they not only work with hypertext documents but also various types of IoT content which requires different techniques and strategies. Second, an IoTSE system might require a more distributed architecture towards a network's edge, depending on the type of query that it resolves. For instances, IoTSE systems that assess queries on the real-time state of the physical world can only achieve low latency by embedding processing capabilities at the edge of a network [2], [3].

As more IoT infrastructures become available, the demand for IoTSE also increases. While a general IoTSE system – the next "Google" – is beyond the horizon due to the vast diversity of IoT content and types of queries for them, specific IoTSE systems that target specific types of query and IoT infrastructure are feasible and beneficial. The exemplary use case, for instance, can be satisfied by an IoTSE system that works with sensors' metadata and readings from a set of

[1]https://www.pwc.com/gx/en/technology/pdf/industrial-internet-of-things.pdf

2640-0715/19/$31.00 ©2019 IEEE
DOI 10.1109/APSEC48747.2019.00039

authorized IoT infrastructures. Therefore, there is a need for a framework to engineering IoTSE systems in a time- and cost-effective manner.

Component-based engineering is a potential solution to the problem of engineering IoTSE systems (Fig. 1). In this approach, IoTSE systems are designed and implemented as a composition of existing components. These components could be developed in-house, reassembled from prior IoTSE systems, or contributed by external developers and researchers. Participants in a component-based IoTSE engineering ecosystem consists of *IoTSE component developers* who contribute their IoTSE-related logics as *IoTSE components*, *IoTSE system developers* who design and compose IoTSE systems from components, and *Architects* who develop reference architectures to guide developers. Given the necessary components and an enabling software infrastructure, IoTSE system developers could engineer a new IoTSE system only by specifying its *composition structure* and *deployment structure*. The prior specifies components to use and how they interact with each other. The latter specifies where each component would be deployed how many replicas are necessary.

The existing body of research across the entire IoTSE systems' workflow could provide components for engineering these systems. However, *two problems* need to be resolved to use these research and engineering efforts as building blocks of new IoTSE systems. *The first problem is architectural: how to design IoTSE components which allow for interoperability, portability, and independent scaling?*

- **Interoperability**: How to enable independently developed IoTSE components to cooperate as per instructions of IoTSE system developers?
- **Portability**: How to enable IoTSE components to operate on a wide range of computing nodes and environment, ranging from clouds to resource-constrained edge devices and from X86-X64 architecture to alternatives such as ARM?
- **Independence**: How to enable individual provisioning of IoTSE components to facilitate fine-grained scaling of IoTSE systems?

The second problem is to limit the complexity, which would emerge from applying a component-based engineering approach. This complexity, if not constrained, would eclipse the benefits brought about by component reuse and composition, rendering component-based engineering impractical.

To address the architectural problem, we propose to design and implement IoTSE components as *containerized services* and specify a set of RESTful APIs for main types of IoTSE components (Section II). To address the complexity problem, we propose a collection of software infrastructure and utilities, collectively known as an *IoTSE Framework*, to hide the complexity of developing and utilizing IoTSE components from developers (Section III).

To demonstrate the feasibility of engineering IoTSE systems from components, we built a reference implementation of the framework and used it to design and develop eight proof-of-concept IoTSE systems for evaluation (Section IV). An analysis on Source Line of Code (SLOC) revealed that the interoperability mechanisms handled by the framework could incur over 90% of the code base of a simple IoTSE system. A comparative study on the performance of the proof-of-concept systems was also conducted to evaluate design decisions of the built IoTSE systems for demonstrating the feasibility of the proposed approach.

II. IoTSE as Containerized Services

To enable component-based IoTSE engineering, the components contributed by different developers must *interoperate* to complete the workflows defined by IoTSE system developers, which could be unanticipated in the development time of the components. The components must also be *portable* to operate on a wide range of computing nodes and environment, ranging from clouds to resource-constrained edge devices and from X86-X64 architecture to alternatives such as ARM. Finally, the components must be *independent* from each other so that the IoTSE systems can provision or remove them individually for scaling. This independence hinges on not only on portability of the components but also on the side-effects of their' interactions. These requirements constitute an architectural problem, whose solution is not algorithmic but design-centric.

To address the stated problem, we propose to design and develop IoTSE components as *containerized services*. This design dictates that components would offer their ability to carry out the main activities of IoTSE systems via explicitly defined service interfaces. Components would be packed with their dependencies into container images, which enable lightweight virtualization that works on both X86/X64 and ARM architecture.

Containerization provides IoTSE components portability. Enforcing inter-component communication via predefined service interfaces offers several advantages which contribute to interoperability and independence of IoTSE components. First, because all of the interactions between IoTSE components must happen explicitly defined service interfaces, they provide a target for standardization, which can lead to interoperability. This standard would explain not only *services that each IoTSE component* offer but also *types and functional scopes of IoTSE components*. Therefore, this standard would also benefit IoTSE component developers and researchers to align and coordinate their effort.

The second advantage of enforcing component interactions to happen via services is minimizing the side-effects of those interactions. This advantage contributes to the independent operation of IoTSE components. Finally, developing IoTSE components as services would create a separation between interfaces and implementations. Such a separation enables the composition of IoTSE systems and facilitate their updates in the future.

While a complete treatment for an IoTSE service interface standard is beyond the scope of this work, we present a simplified RESTful API to clarify the introduced concepts

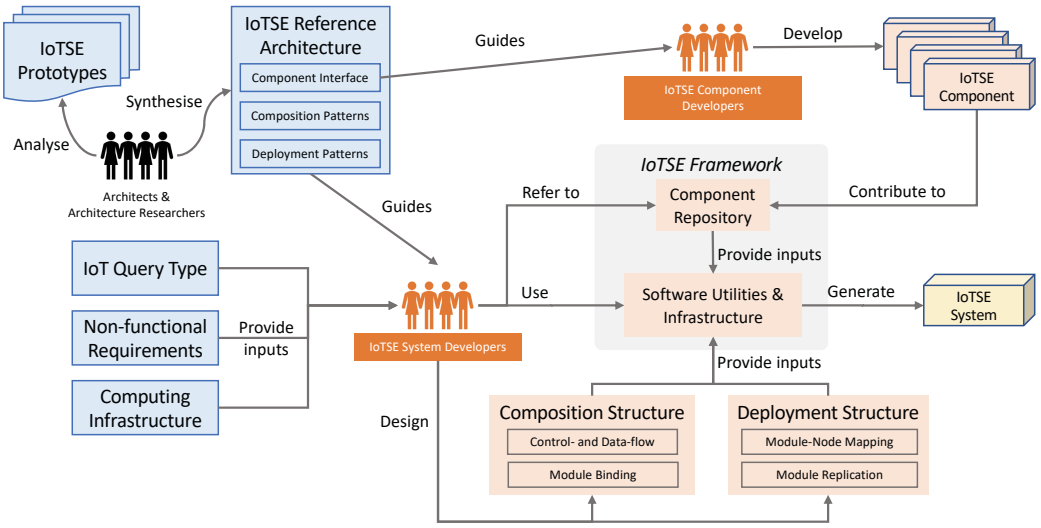

Fig. 1. A component-based approach to engineering IoTSE system.

(Table I). These service interfaces would be used in the proof-of-concept evaluation (Section IV). To support asynchronous processing of IoTSE components, each POST interface returns a unique identity for every request. This identity is then used for tracking and getting the processing results.

The component types presented in this set of service interfaces are identified based on an IoTSE's key activities, which we have taken from a paper [1]. *Detector* and *Collector* components are responsible for finding and gathering IoT content within an IoT infrastructure that an IoTSE system may target. *Storage*, *Indexer*, and *Searcher* components are responsible for holding the collected IoT content and resolving queries on them. *Aggregator* component combines search results from multiple Searcher components. Finally, *Facade* component provides a single entry point to simplify interactions with an IoTSE system.

It should be highlighted that we do not mandate RESTful API over HTTP as the only mean to interact with components.

III. A Framework for Component-based IoTSE Engineering

Engineering IoTSE components as containerized services would inevitably increase the complexity of both development and utilization. To a certain degree, the incurred complexity would eclipse the benefits brought about by an architectural solution, rendering it unusable. This paper proposes and designs a collection of software components, infrastructure, and utilities, collectively defined as a *Framework for Component-based IoTSE Engineering* to handle the complexities of component-based engineering on behalf of software developers (Fig. 2)). This section introduces the design and purposes of the components that can make up the framework of engineering IoTSE. A reference instantiation of the framework would be discussed as a part of a proof-of-concept evaluation in Section IV.

A. IoTSE Component Kernels

The priority of an IoTSE component developer would be the component's logic such as detecting URLs of sensors on the Web, not learning and developing a service interface for the component. The challenge, therefore, is to make the task of developing interoperable containerized services from the component's logics effortless and invisible.

The *IoTSE Component Kernels* of the Framework address this challenge. They act as black boxes which transform service requests into inputs of components' logics and transforms their outputs into service responses. For *each component type* specified in Section II, *one IoTSE component kernel* exists. Each kernel contains an implementation of the service interface of its corresponding component type as well as the necessary infrastructure to host it such as a Web server. Kernels also contain software clients for interacting with the service registry and orchestration engine which would be used by system developers when composing new IoTSE systems.

Given a set of kernels, developing IoTSE components as containerized services compliant with the interfaces in Table I incurs only three additional activities:

- *Acquiring a suitable kernel:* This decision is based on the correspondence between the developed component's logic and the defined component types. The kernel itself could

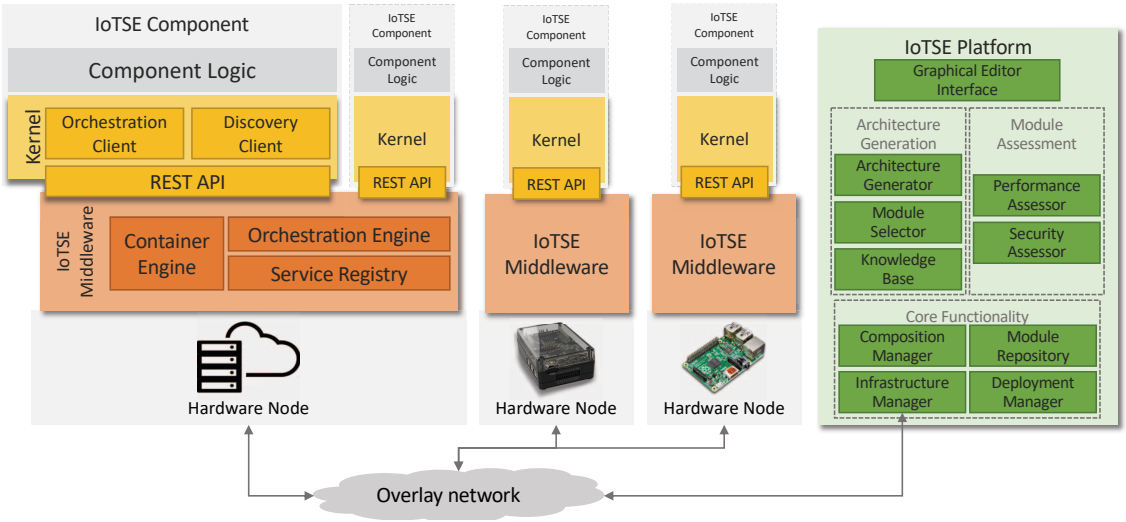

Fig. 2. A Framework for IoTSE Development. Each hardware node hosts an IoTSE middleware instance, which run a number of IoTSE components.

be acquired via a Web portal or a package manager. Both of which approaches would be familiar to developers.

- *Building a connector between the component logic and the pre-built service interface.* It would translate and route the interface's inputs to the logic's inputs, and vice versa with the outputs. While the connector introduces complexity, it offers component developers more control, making it an acceptable trade-off.
- *Containerize the component and publish its image.* Developers could carry out this task with a wide range of existing software utilities, such as Docker CLI [2] and Kitematic[3].

B. IoTSE Middleware and Platform

Component-based engineering of IoTSE systems also incur overheads in terms of providing an infrastructure for operating component services and of controlling this infrastructure to provision IoTSE components on computing nodes and start the whole system. An ideal outcome from the perspective of IoTSE system developers would be *zero-code composition* which denotes the ability to compose IoTSE systems without reprogramming the existing components, nor developing additional software components. In the proposed IoTSE Framework, these challenges are met by the *IoTSE Middleware* and *IoTSE Platform*.

1) Middleware: The IoTSE Middleware provides an infrastructure to *run IoTSE components* on a computing infrastructure and *control the flow of data and control* among the deployed components to compose an IoTSE system. On each computing node of an IoTSE system, software developers would use an instance of the middleware. These instances cooperate to maintain an overlay network as a backbone of an IoTSE system, allowing its components to exchange data and requests across different computing nodes. Each middleware instance contains a container engine which is compatible with the container image format used by component developers to run IoTSE components on computing nodes.

A subset of middleware instances also contains a service orchestration engine and a service registry. The role of the orchestration engine (e.g., Conductor[4]) is to control service invocations and routing of responses according to workflows specified by software developers. *The kernels and the middleware are paired to ensure interoperability*. It means that service discovery and orchestration clients embedded in the kernels are matching with the registry and orchestration engine embedded in the middleware.

2) Platform: While the middleware by itself offers an adequate capability to run and orchestrate IoTSE components, the process of provisioning an IoTSE system would be still repetitive and error-prone due to the decentralization of the computing nodes. Moreover, responsibilities of the middleware do not cover the features which simplify the composition and management of IoTSE systems such as automatic service selection and profilers.

The IoTSE Platform of the Framework addresses these issues. It interfaces with authorized IoTSE middleware instances to enable centralized provisioning and control of IoTSE systems. It also hosts utilities which support system developers in designing and composing IoTSE systems. The platform can be hosted on cloud-based infrastructure and shared among IoTSE system developers. Alternatively, developers can host private instances of the platform.

The *Core Functionality Group* monitors computing nodes, provisions IoTSE components, and control their interactions. The *IoT Infrastructure Manager* interacts with IoTSE middle-

[2]https://github.com/docker/cli
[3]https://kitematic.com
[4]https://github.com/Netflix/conductor

TABLE I
COMPONENT TYPES AND THEIR REST INTERFACES.

Component	URL Endpoint	HTTP Verb	Functionality
Detector	/api/new-res-ids	GET	Start content detection process and return URL of detected IoT content
Collector	/api/res-contents	POST	Invoke content collection on the given set of URL and return a req-id for future retrieval collected data.
	/api/res-contents/req-id	GET	Get the set of collected IoT content identified by req-id
Storage	/api/iot-resources	POST	Store set of IoT content in the storage for future retrieval
	/api/iot-resources/res-id	GET	Retrieve the IoT content identified by res-id
Indexer	/api/index	POST	Invoke the indexing mechanism
Searcher	/api/queries	POST	Submit a query; invoke the query processing; creates a res_id
	/api/results/res-id	GET	Retrieve a set of search results identified by res-id
Aggregator	/api/agg-results	POST	Accept a list of search results for future aggregation
	/api/agg-results/res-id	GET	Aggregate result sets linked to the res-id
Facade	/queries	POST	Submit a query; initialise query processing workflows; creates a result_id for future lookup.
	/results/result_id	GET	Retrieve a set of search results identified by result_id

ware instances to manage hardware nodes of an IoTSE system. The *Deployment Manager* uses the IoTSE infrastructure manager to place IoTSE components on the hardware nodes. The *Composition Manager* interacts with orchestration engines on IoTSE middleware instances to orchestrate the control- and data-flow among IoTSE components. The *Component Repository* maintains addresses or images of the available IoTSE components.

The *Architecture Generation group* contains the utilities that support the design of composition structure and deployment structure and the binding of components into those structures. These structures can be specified via graphical diagrams or structured textual documents such as JSON. The *Component Assessment group* contains utilities to conduct automated tests of the available components.

Given the middleware and access to the IoTSE platform, engineering an IoTSE system would involve the following activities:

- Provision middleware instances on the computing nodes to be used in an IoTSE system.
- Design a composition and a deployment structure of an IoTSE system.
- Binding specific IoTSE components into a composition structure.
- Providing the structure and binding specifications to the IoTSE platform to compose an IoTSE system.

IV. PROOF-OF-CONCEPT EVALUATION

To assess the feasibility of component-based IoTSE engineering with the proposed framework, we have developed and evaluated eight proof-of-concepts IoTSE systems. All of the proof-of-concept IoTSE systems were composed of eight IoTSE components which have been developed based on a reference implementation of our IoTSE framework. The sensor sources for these proof-of-concept systems are our in-house developed IoT gateways which expose both real sensors and replayed sensory data as Web resources following OGC's SensorThing API standard[5].

Two of the proof-of-concept systems query only meta-data, two query only sensor readings, and the remaining four work with both types of content. These complex systems were built from the components of simpler ones to resolve queries for Web-enabled sensors based on both their static meta-data and real-time readings. For instance, they can find "sensors which measure temperature in Celsius, whose latest reading is less than 25 degrees". The source code of components and configurations of the proof-of-concept systems are available on our repository[6]

The functioning of these proof-of-concept systems demonstrates the feasibility of a component-based IoTSE with the architecture and software framework proposed in this paper. To assess the benefit of the framework on the complexity of components, we compare the size difference in Source Line of Code (SLOC) between component logics and interoperability mechanisms handled by the kernels. We have also conducted a comparative study on the performance of the proof-of-concept systems to demonstrate the ability to conduct experiments on design alternatives of IoTSE systems, which was facilitated by the zero-code composition ability offered by the framework.

A. A Reference Implementation of the Framework

1) Component Kernels.: We built the RESTful API shown in Table I using Flask RESTful library [7]. We implemented URL endpoints as Python classes and HTTP verbs as methods of these classes. These methods process the incoming requests and pass them to the connectors which would be supplied by component developers. The kernel includes an instance of the Gunicorn Web Server[8] to host this API. The kernel also contains a Netflix's Conductor client for service orchestration purposes. A set of Python- and Bash-based utilities were included to support the compilation of classes of a component to form a single deployment unit and generate a container image. The total size of the kernel is *1189 Source Lines of Code (SLOC) (961 SLOC sans the Conductor client).*

[5]http://www.opengeospatial.org/standards/sensorthings
[6]Removed for double-blind review.
[7]https://flask-restful.readthedocs.io/
[8]https://gunicorn.org/

2) Middleware and Platform.: The reference middleware was built upon Docker Engine and Conductor for managing containers and orchestrating component services, respectively. All Docker Engine instances were configured to form a swarm. The reference platform consist of Python- and Bash-based utilities. They interact with Docker Engine instances and Conductor engines to compose and deploy components. System developers control these tools with two JSON documents, which specify a deployment structure and a composition structure.

B. IoTSE Components and Effort Reduction

The proof-of-concept IoTSE systems discover, index, and resolve queries on both metadata and real-time sensor readings. Eight IoTSE components handle these activities. The tasks of detecting Web-based sources of sensory data and sensors within those sources are controlled by *Source Detector* and *Sensor Detector* components, both of which belong to the Detector component type. The prior contains a predefined list of URL of the deployed IoT gateways. The latter leverages OGC's SensorThing API standard, with which of the gateways are compliant, to query and construct a set of URLs directing to sensors' representation and readings.

The tasks of collecting, indexing, and resolving queries on meta-data and sensor readings are handled by two sets of components: Meta-data Collector and Searcher, and Reading Collector and Searcher. Both Collector components are customized HTTP clients for extracting meta-data or sensor readings from raw JSON documents retrieved from the gateways. Both Searcher components combine the functionality and services of Storage, Indexer, and Searcher component types. Internally, they maintain and control internal MongoDB instances. While aggregating component types is not recommended in a production environment, this decision simplified the proof-of-concept IoTSE systems and helped us to highlight the key features of the framework more easily. The last two components are Aggregator and Facade. Their functionality matches the component type from which they instantiated.

The SLOC of eight components in comparison to their kernels are shown in Table II. We used the Source Line of Code (SLOC) as a proxy metric for the amount of development effort in each component. We utilised `pygount`[9] – a Python-based utility – to analyse components' source code and measure SLOC. Reduced effort was calculated as the ratio between SLOC of logic and total SLOC of a component comprising both logic and kernel codes. The gap between SLOC of the kernel and the components' logics, reaching at least 92% of the total size of a component, highlights the amount of effort that the use of a kernel could potentially reduce for IoTSE component developers.

It should be noted that this result is not to conclude that such an outcome could always be achieved in a production environment where component logic would be substantially more complex. Instead, it highlights that when IoTSE components are relatively simple, as in the case of the proof-of-concept systems, the complexity added by the containerized service design of IoTSE components could eclipse the effort necessary to develop the components' logics by multiple folds. Therefore, component developers would not be likely to adopt a component-based IoTSE engineering approach without the support from the proposed kernel.

[9]https://pypi.org/project/pygount/

TABLE II
SOURCE LINES OF CODE (SLOC) BREAKDOWN OF EIGHT COMPONENTS.

	Kernel (w/ Utilities) + Logic	Kernel (w/o Utilities) + Logic	Logic	Reduced Effort
Source Detector	1200	972	11	93.7%
Sensor Detector	1249	1021	60	98.8%
Metadata Collector	1239	1011	50	94.8%
Metadata Searcher	1246	1018	57	94%
Reading Collector	1239	1011	50	94.8%
Reading Searcher	1250	1022	61	93.6%
Aggregator	1265	1037	76	92%
Facade	1215	987	26	97.3%
All Components	9903	8079	391	**96%**

C. Proof-of-concept Systems and Experiment

From the developed components, we composed and deployed eight IoTSE systems using the reference IoTSE middleware and platform. Two of the proof-of-concept systems query only meta-data and therefore comprise only Meta-data-related components. The two following proof-of-concept systems use Reading-related components to query sensor readings exclusively. The last four systems utilize all eight components.

The variation among these proof-of-concept systems lies in their composition structures. First, they vary in the timing between the discovery of IoT content and the assessment of queries. In the Parallel Discovery (PD) structure, content discovery runs in parallel to the query assessment. In the Interlaced Discovery (ID), the content discovery runs only when a query arrives. Second, they differ in the timing between the ranking of meta-data and sensor readings when responding to a query. In Parallel Search (PS), meta-data- and content-based search run in parallel. In Sequential Search (SS), they run in a sequence. Putting these notations together, a PD-PS structure means that an IoTSE crawls IoT content in parallel to the query assessment, and the evaluation on meta-data and sensor readings are carried out in parallel.

The zero-code composition ability offered by the proposed framework provides several benefits, including the ability to evaluate and compare alternative design decisions experimentally. For instance, let's assume that an IoTSE system development team is deciding on the composition structure of an IoTSE system. Their assumptions include PD would yield

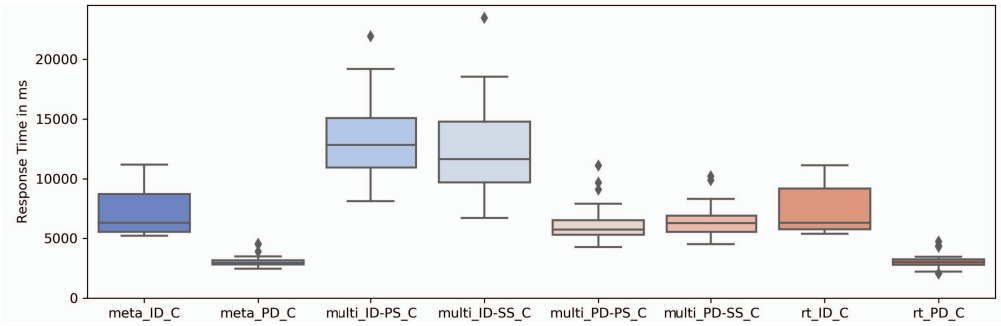

Fig. 3. The query response time of IoTSE prototypes, measured in milliseconds.

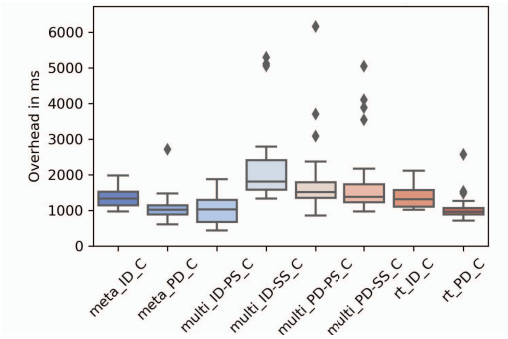

Fig. 4. The overhead caused by the middleware on IoTSE prototypes, measured in milliseconds.

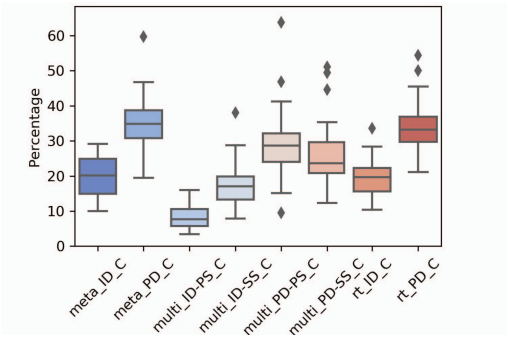

Fig. 5. The overhead caused by the middleware on IoTSE prototypes, measured in percentages of the response time.

better performance than ID, and PS would produce a substantial performance gain over SS. Instead of relying solely on reasoning, the zero-code composition ability allows the team to engineer some prototypes that capture their alternative design decisions and conduct experiment with those prototypes, with minimal effort required.

To demonstrate this point, we have conducted the stated experiment on the proof-of-concept systems. We quantified the performance of an IoTSE system via its response time. This metric is measured as the delay between the submission of a query and the response. We used IntelLab[10] data set for this experiment. This dataset records topology information, humidity, temperature, light, and voltage values. Fifty-four sensors in the Intel Berkeley Research lab recorded these values between 28th February 28th and April 5th, 2004. We replayed the data set to IoTSE prototypes with an in-house developed IoT gateway. We then queried the prototypes for *sensors that measure temperature in Celsius, whose latest reading is less than 25*. A python script which runs on the client-side measures the response time. We repeated the experiment 30 times for each prototype. We conducted the experiment on an Intel Xeon E3 workstation with 8GB of memory.

The results of the experiment (Figure 3, 4 and 5) confirm the assumption that PD offers a better performance than ID, and so is the case with PS and SS. However, PS did not yield a substantial performance gain as the prediction in the hypothetical scenario. Such an experiment-backed insight on design decisions would be more difficult to achieve without the zero-code composition ability offered by the proposed framework.

[10]http://db.csail.mit.edu/labdata/labdata.html

V. Related Work

Our work approaches IoTSE from a perspective which has been relatively unexplored in the existing IoTSE literature. We investigate the architectural support and the tools to weave branches of IoTSE together. Meanwhile, the majority of IoTSE literature has focused on technical issues pertaining a branch of IoTSE, for example, real-time sensor search (Dyser [2], CSS [3]), context-based sensor search (CASSARAM [4], ThingSeek [5]), functionality search based on semantics [6], [7], object localization (Snoogle [8], MAX [9], OCH [10]), and discovery services based on EPCglobal specifications [11].

As a result, there are only a few pieces of work in the IoTSE literature that are related and directly comparable to our work presented in this paper. These associated works model IoTSE components as shared software libraries and offer templates to simplify their development. For example, ThingSeek [5] focuses on detecting URL of sources and sensors, while Kernel-based IoTSE [12] covers the entire workflow of an

IoTSE instance. These efforts share two limitations. First, developing an IoTSE system using these approaches requires deep integration of components into its codebase. This integration impedes the modification and independent scaling of components. Second, the shared libraries make assumptions on how an IoTSE component should be implemented (i.e., variables to use, functions to implements).

Our solution reported in this paper addresses both of these limitations. By modeling IoTSE instances as a workflow-based composition of containerized Web services, our solution simplifies the modification of an IoTSE system and allows independent scaling of its components. Moreover, our solution gives component developers complete control over data models, algorithms, and technologies utilized for implementing the components of an IoTSE system.

VI. Conclusion

As the investment in IoT infrastructure expands towards $832 billion in 2020, the demand for IoTSE systems also increases. A component-based approach could provide a time- and cost-effective solution to developing reliable and flexible IoTSE systems. Two problems must be addressed to realize a component-based IoTSE engineering approach: developing an architecture for IoTSE systems which can help maximize the interoperability, portability, and independence of their components, and isolating the complexity introduced by this architectural solution to component and system developers.

This paper proposed to design and implement IoTSE components as containerized services and introduced a software framework to simplify the process of developing and utilizing these components. The evaluation of the proof-of-concept systems has demonstrated the feasibility of composing functional IoTSE systems from independently produced components and of combining the elements of multiple IoTSE systems into a more capable IoTSE system. The evaluation has also highlighted that the complexity handled transparently by the IoTSE framework could account for over 90% of the code base of a simple IoTSE component. The evaluation has also demonstrated the feasibility of the zero-code IoTSE composition to support IoTSE system developers in making more informed design decisions via experimentation. However, the decomposition of monolithic IoTSE systems into independently developed services has also increased the overhead of the coordination between components, which slowed down their responses. A potential future work, therefore, is optimizing service interactions and orchestrations within IoTSE systems to retain both performance and modularity.

For future work, a potential direction would be to tighten the security over IoTSE components to prevent possible data leakage by introducing additional validation and enforcement mechanisms into the IoTSE kernel. Another direction can be to enhance the performance of the IoTSE middleware by adding caching, load balancing, and alternative service invocation model such as publish-subscribe. Finally, additional automation could be introduced to the IoTSE platform to simplify the composition process further. For instance, Semantic- and Quality-of-Service-based component selection mechanisms could be added to automate the component-binding step, and self-adaptive mechanisms such as MAPE-K model could give IoTSE systems the ability to reconfigure itself in response to incoming queries and infrastructure changes.

References

[1] N. K. Tran, Q. Z. Sheng, M. A. Babar, and L. Yao, "Searching the web of things: State of the art, challenges, and solutions," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, p. 55, 2017.

[2] B. Ostermaier, K. Romer, F. Mattern, M. Fahrmair, and W. Kellerer, "a Real-Time Search Engine for the Web of Things," in *Proceedings of the 1st International Conference on the Internet of Things (IOT)*. IEEE, 2010, Conference Proceedings, pp. 1–8.

[3] C. Truong and K. Römer, "Content-Based Sensor Search for the Web of Things," in *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2013, Conference Proceedings, pp. 2654–2660.

[4] C. Perera, A. Zaslavsky, C. H. Liu, M. Compton, P. Christen, and D. Georgakopoulos, "Sensor Search Techniques for Sensing as a Service Architecture for the Internet of Things," *IEEE Sensors Journal*, vol. 14, no. 2, pp. 406–420, 2014.

[5] A. Shemshadi, Q. Z. Sheng, and Y. Qin, "Thingseek: A crawler and search engine for the internet of things," in *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2016, Conference Proceedings, pp. 1149–1152.

[6] B. Christophe, V. Verdot, and V. Toubiana, "Searching The'web of Things'," in *Proceedings of the 5th IEEE International Conference on Semantic Computing (ICSC)*. IEEE, 2011, Conference Proceedings, pp. 308–315.

[7] M. Mrissa, L. Médini, and J.-P. Jamont, "Semantic Discovery and Invocation of Functionalities for the Web of Things," in *Proceedings of the 23rd IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE, 2011, Conference Proceedings, pp. 281–286.

[8] H. Wang, C. C. Tan, and Q. Li, "Snoogle: A Search Engine for Pervasive Environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 8, pp. 1188 – 1202, 2010.

[9] K.-K. Yap, V. Srinivasan, and M. Motani, "Max: Human-Centric Search of the Physical World," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys)*. ACM, 2005, Conference Proceedings, pp. 166–179.

[10] C. Frank, P. Bolliger, F. Mattern, and W. Kellerer, "the Sensor Internet at Work: Locating Everyday Items Using Mobile Phones," *Pervasive and Mobile Computing*, vol. 4, no. 3, pp. 421–447, 2008.

[11] S. Evdokimov, B. Fabian, S. Kunz, and N. Schoenemann, "Comparison of discovery service architectures for the internet of things," in *Proceedings of the 2010 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC)*. IEEE, 2010, Conference Proceedings, pp. 237–244.

[12] N. K. Tran, Q. Z. Sheng, M. A. Babar, and L. Yao, "A kernel-based approach to developing adaptable and reusable sensor retrieval systems for the web of things," in *Proceedings of the 18th International Conference on Web Information Systems Engineering*. Springer International Publishing, 2017, Conference Proceedings, pp. 315–329.