SeeQ: A Programming Model for Portable Data-driven Building Applications

Dimitris Mavrokapnidis
d.mavrokapnidis@ucl.ac.uk
University College London
London, UK

Gabe Fierro
gtfierro@mines.edu
Colorado School of Mines
National Renewable Energy Laboratory
Golden, Colorado, U.S.A.

Maria Husmann
maria.husmann@siemens.com
Siemens AG
Zug, Switzerland

Ivan Korolija
i.korolija@ucl.ac.uk
University College London
London, UK

Dimitrios Rovas
d.rovas@ucl.ac.uk
University College London
London, UK

ABSTRACT
This paper introduces SeeQ, a programming model and an abstraction framework that facilitates the development of portable data-driven building applications. Data-driven approaches can provide insights into building operations and guide decision-making to achieve operational objectives. Yet the configuration of such applications per building requires extensive effort and tacit knowledge.

In SeeQ, we propose a portable programming model and build a software system that enables self-configuration and execution across diverse buildings. The configuration of each building is captured in a unified data model — in this paper, we work with the Brick ontology without loss of generality. SeeQ focuses on the distinction between the application logic and the configuration of an application against building-specific data inputs and systems. We test the proposed approach by configuring and deploying a diverse range of applications across five heterogeneous real-world buildings. The analysis shows the potential of SeeQ to significantly reduce the efforts associated with the delivery of building analytics.

CCS CONCEPTS
• Software and its engineering → Abstraction, modeling and modularity; • Information systems → Graph-based database models.

KEYWORDS
Programming, Analytics, Portability, Scalability, Brick, RDF, SHACL, Metadata, Semantic Web, Ontologies

ACM Reference Format:

1 INTRODUCTION
The potential for data-driven services to provide insights is increasingly being understood in the built environment context. When considering buildings in particular, a range of data-driven building applications — covering broad domains in Fault Detection and Diagnostics (FDD), Predictive Controls, and Building-to-Grid, — have supported outcomes like improved operation, better environmental comfort, more sustainable and cost-effective operation [12]. This is particularly true in commercial buildings with more sophisticated sensing and control infrastructure and ownership structures.

Commercial building floor area is projected to rise from 24 to 57 bn m², with an annual cooling and heating energy demand rising to more than 1050 TWh by 2050 [30]. Understanding operational inefficiencies and improving how buildings are operated and maintained is critical to ensuring economical energy use and creating spaces that meet end users’ needs and well-being. Data-driven applications can help manage the complexity associated with building operations, and provide data-driven insights, to support building oversight and management. While technological infrastructure is available, adopting such approaches at scale is lagging due to the repetitive efforts and tacit knowledge to re-configure and deploy them across heterogeneous buildings and sites.

Configuring data-driven applications requires a good understanding of the building set-up, including the building heating, cooling and refrigeration systems, ventilation and electrical systems, and building automation and control systems. In many cases, the required information is sparse, out-of-date, or distributed in multiple locations; this requires discovering and accessing data from diverse data sources [16] including as-designed information like Building Information Models (BIM) [27], handover information like COBie drops, and operational information extracted from the Building Management Systems (BMS) [23]. The challenge of discovering building information, combined with the poor (or non-existent) state of documentation of many BMS systems, results in challenging deployments of such approaches in most buildings [25]. The result is that many applications are still developed on an ad-hoc basis and are hardly reusable across buildings [12, 25].
The ability to capture information about physical, logical and virtual building assets and their relationships has been the subject of extensive research. As a result, several metadata models such as Brick [7], Project Haystack [5], Real Estate Core [20], and ASHRAE 223P [13] have been proposed to make data easily discoverable and accessible through uniform machine-readable representations. Recent work demonstrates that leveraging such representations can make intelligent building software wholly or partially self-configuring [17, 21]. Still, the learning curve is quite steep, requiring developers to be familiar with several technologies: RDF[3], OWL[1], SPARQL[2], and so on. Nevertheless, despite the potential of those advancements in expressing building heterogeneity, we still lack a universal paradigm for developing applications and encapsulating information requirements so that they can be developed once and executed across multiple buildings.

This paper introduces SeeQ, a portable programming model aspiring to shift the configuration burden from the deployment stage to the application authoring phase. In particular, we separate the expression and execution of data-driven applications from how those are configured for specific buildings. Core to SeeQ is the notion of computational quantities (CQs). These are reusable logical identifiers for real-world quantities that abstract away a set of possible configurations. They allow applications to be expressed clearly in terms of familiar quantities (e.g. “mixed air temperature”) while hiding the complexity of delivering those quantities across buildings. We show that CQs enhance the reusability of building software, making it portable and thus cheaper to develop and deploy.

We summarise the contributions of this work as follows:

1. We identify four main application portability challenges, as discovered within the existing industrial and research efforts,
2. formally define a programming model for the development of portable building applications,
3. architect a software system adopting the principles of the proposed programming model,
4. demonstrate how a portable application is authored and reused across heterogeneous buildings, and
5. evaluate SeeQ, drawing insights from the development of a broad family of data-driven building applications and their deployment across 76,100 m² of commercial building space.

2 BACKGROUND AND PRIOR WORK

Portability refers to a building application’s ability to adapt its execution to a given environment without manual intervention or configuration [15]. We address three areas of existing work in enabling building application portability: (1) building metadata models, (2) application platforms, and (3) portability mechanisms.

2.1 Building Metadata Models

Past work has established the difficulty of writing data-driven building applications due to a lack of standard digital representations that facilitate data discovery [10, 11]. Contemporary research develops standardized representations that address this lack of introspection and discoverability, including Project Haystack [5], Brick Schema [7], RealEstateCore [20], and others [13, 28, 29].

In these models, information about physical assets like HVAC, lighting, electrical and plumbing systems and their interconnection is represented by directed graph structures. These graphs also encode the identity of data sources, building assets, and their relationships, providing a unified representation of the smart building as a cyber-physical system of systems. Applications query these graphs to configure their operation; this involves retrieving the composition and topology of building systems and identifying data sources or control inputs for the application.

Despite the success of these representations, accessing the required data for specific applications remains challenging, mainly because developers must be familiar with the structure and content of the graph as well as the query language required to retrieve metadata [9]. For example, Bhattacharya et al. reported an inability to run three simple diagnostics applications in a portfolio of 10 buildings due to a lack of the required semantic richness [10].

2.2 Application Platforms

Deploying building analytics relies on the availability of application middleware platforms. These platforms support data storage and management, access control, and provide uniform APIs for applications. Recent building middleware platforms recognise the challenge of managing the lifecycle of hundreds or thousands of instances of such applications and offer several methods to enable the mass-configuration of applications, often through semantic metadata models or tagging schemes [7].

Building Application Stack (BAS) [24] and the later BOSS [14] provide a fuzzy query interface over a graph of building components and control interfaces, enabling application authors to instantiate a graph of system-agnostic objects which describes the various building entities and their relationships. However, the lack of a formal data model limits the representation of various building system configurations. BuildingDepot [32] adopts a template-based approach which restricts user applications to those that can be expressed using pre-defined sets of building entities and data sources. These templates simplify development but their construction requires manual mapping of building data sources.

Recent application platforms build on structured semantic metadata to simplify configuration. Mortar [17] uses queries over Brick models to capture flexibility, offloading the burden of portability onto the developer. Capturing this variability is difficult: the simple set of Measurement & Verification (M&V), FDD, and sensing applications deployed over the Mortar data set only ran on between 2–65% of the buildings considered. Sky Foundry [6] and Energon [21] use non-standard and purpose-built programming and query languages to reduce lines of code and development complexity but still require developer-driven reconfiguration between deployment sites. Bemani et al. proposed a query relaxation algorithm to improve the retrieval of building data through SPARQL results, increasing query portability across different building configurations.

These approaches often address only the relationship between descriptions of the building and the actual telemetry, leaving the implementation of application portability to the developer. Mavrokapnidis et al. propose a portable programming model that simplifies the development of these applications but focuses only on fault detection rules. Our work significantly enhances and generalizes this approach to a broader family of portable building applications.
2.3 Portability Mechanisms
Portability mechanisms promise to enhance data discoverability and therefore increase the ability of one application to be reused across multiple buildings. Energon [21] introduced a new SQL-like querying language that allows the expression of graph queries without the need to know the internal structure of Brick models. Though Energon promises to reduce the lines of code required to retrieve metadata, developers must become familiar with a new querying language, i.e. EnergonQL.

Nevertheless, “porting” a building application requires ensuring sufficient data quality. For this reason, Fierro et al. introduced BuildingMOTIF [18], a feedback mechanism that enables the creation of “semantically sufficient” models to meet the data requirements for a particular application. By normalizing the structure of metadata models, BuildingMOTIF can help reduce the degree of portability required for an application by bounding the set of configurations that the application must understand.

Many pieces of portable application solutions for smart buildings have been established in the literature. However, these components have yet to be bound into a single solution that properly addresses key application portability challenges while providing a familiar and accessible abstraction to the developer.

3 APPLICATION PORTABILITY CHALLENGES
We identify four main challenges currently hindering the development of portable applications, and illustrate them throughout the paper, using the following running examples from two widely adopted open-source building analytics libraries:

Running Example 1: Rule $R_1$ from the Air-Handling Unit (AHU) Performance Assessment Rules (APAR) [31] is a typical example of a rule-based Fault Detection analytic that identifies abnormal operation of a single-duct AHU during heating mode (i.e., Heating Coil Valve: $H_{c_{\text{cop}}} > 0$). According to APAR’s documentation, $R_1$ verifies that AHU’s supply air temperature ($T_{\text{ma}}$) is greater than the mixed air temperature ($T_{\text{ma}}$) plus the temperature drop over the supply fan $\Delta T_{\text{sf}}$ minus a threshold $\epsilon$. The following inequality expresses $R_1$:

$$R_1 : T_{\text{sa}} < T_{\text{ma}} + \Delta T_{\text{sf}} - \epsilon, \text{ while } H_{c_{\text{cop}}} > 0. \quad (1)$$

Running Example 2: $D_{\text{mp_{leak}}}$ from ASHRAE’s Guideline-36 is an application that detects a leaking damper in VAV units with reheat. In particular, the application generates an alarm when the damper position ($P_{\text{pos_{dmp}}}$) is 0%, and the airflow sensor ($F_{\text{sa}}$) reading is above the larger of either 10% of the cooling maximum airflow setpoint ($F_{\text{sp}_{\text{clg}}}$) or 50 cfm for 10 minutes while the fan ($F_{\text{fan}_{\text{stat}}}$) serving the zone is “ON”. We can express this application as:

$$D_{\text{mp_{leak}}} : \begin{cases} P_{\text{pos_{dmp}}} = 0, \\ F_{\text{sa}} > \max(0.1 \cdot F_{\text{sp}_{\text{clg}}}, 50), \\ F_{\text{fan}_{\text{stat}}} = "ON" \end{cases}, \text{ for 10 minutes.} \quad (2)$$

3.1 Model expressivity (C1)
Accessing the required data points from a graph is an expert-dependent task that requires familiarity with the query language and the content of a graph model to extract the required information. Configuring and executing our running example $R_1$ as expressed by Eq. (1) over a Brick model requires the presence of a brick: Mixed_Air_Temperature_Sensor point (i.e. $T_{\text{ma}}$) and a brick: Supply_Air_Temperature_Sensor point (i.e. $T_{\text{sa}}$) across every AHU in a building. A point may be modelled in a few different ways. For example, a mixed air temperature sensor is usually associated with an AHU directly but may also be modelled on a component of the AHU.

3.2 Data Availability (C2)
The lack of available building data is a major barrier to reproducing a building application across different buildings. For example, running $R_1$ requires the Mixed Air Temperature (i.e. $T_{\text{ma}}$) across different AHUs. Yet, $T_{\text{ma}}$ is not always available as a sensor point. Instead, if a sensor is missing, developers can possibly use other data points to estimate this quantity. For example, it may be estimated as: $T_{\text{ma}} = T_{\text{oa}} + D_{\text{oa}} + T_{\text{ra}} + D_{\text{ra}}$ where $T_{\text{oa}}/T_{\text{ra}}$ & $D_{\text{oa}}/D_{\text{ra}}$ stand for the outside/return air temperature & damper position respectively.

3.3 System applicability (C3)
To configure and execute our $R_1$ and $D_{\text{mp_{leak}}}$ examples, the developer must respectively determine not only the available AHUs and dampers in a building but also their system configuration. In particular, $R_1$ can only be implemented in single-duct AHUs while $D_{\text{mp_{leak}}}$ requires the presence of VAV units with reheating capability. Extracting this information from a Brick model requires expertise and familiarity with a query language to write and execute multiple and often complex queries across different buildings.

3.4 Temporal Configuration (C4)
Even if all the required data points are available, most implementations of building applications are not portable due to a lack of descriptions of their temporal requirements. In our running example, $D_{\text{mp_{leak}}}$ raises an alarm when all three clauses of eq. 2 is true for 10 minutes continuously, suggesting a 5-min rolling average window with a 1-min sampling period for every point value [19]. Even if all point values are available, the developer must manually fetch, clean, process and aggregate data to meet these requirements. This manual temporal configuration process presents an additional burden in porting a building application.

4 PORTABLE PROGRAMMING MODEL
In this section, we present the formal design of our programming model for expressing portable building applications which can self-customize their operation based on properties of the building in which they are deployed. Portable applications are constructed using logical identifiers (termed computational quantities or CQs), which abstract away the identification and computation of quantities used in the application’s logic. These logical identifiers are resolved to numerical values before execution of an application by a special compilation step which performs the necessary porting of the application to a particular building.

We first formalize the definition of CQs and detail three types of CQs used to build portable applications: GraphCQs, VirtualCQs, and DefaultCQs. We then formally define portable applications in terms of CQs and explain the resolution process by which an application is “ported” to execute on a given building described by
4.1 Semantic Metadata Background

We briefly cover a few semantic metadata concepts used in defining computational quantities. Our application portability mechanism depends on an RDF-based [3] graph representing a building built with the Brick ontology. This graph is called the semantic metadata model of a building.

An RDF graph consists of a set of node-edge-node tuples called triples. The labels of the nodes and edges refer to entities (physical, virtual or logical “things”) present in the building and concepts and relationships defined in the Brick ontology [7]. This data structure makes it possible for tools to reason about the availability and configuration of data sources and other entities in a building. The Brick ontology encodes additional contextual information that makes it possible to identify entities by their behavior and properties rather than by name. This abstraction is key for application portability.

Our proposed mechanism also uses the SHACL constraint language [4]. We elide a full description of SHACL due to space constraints. SHACL permits the definition of shapes, which are groups of constraints and conditions over graphs. Shapes contain a specification (target) of which parts of a metadata graph they apply to. A SHACL validation engine interprets shapes with respect to a metadata graph. It reports whether or not all of the constraints and conditions in a shape were satisfied on each part of the graph where the shape applies.

4.2 Computational Quantities

A computational quantity (CQ) is a portable definition of a physical quantity that can be used in portable applications. In their implementation, applications refer to CQs using logical identifiers (e.g. \( T_{ma} \) for mixed air temperature). CQs are portable because they encode multiple ways that a quantity may be found or derived in a building. Quantities may be (a) observed directly by sensors or exposed directly by digital registers, (b) observed indirectly through computation or extrapolation from other observations, or (c) assumed to be a default value.

4.2.1 Motivating Example. We first present an intuitive explanation of how CQs work before formally defining their operation below. Consider a CQ representing the mixed air temperature for our running example of performing fault detection (§3) on an air handling unit (Figure 1). A portable definition of mixed air temperature (\( T_{ma} \)) must be able to handle any subset of the data sources (indicated by circles) being present in the building. \( T_{ma} \) might be observed directly (via the Mixed Air Temperature Sensor); however, if this is not available, then it is possible to estimate the mixed temperature using the average of the outside and return air temperatures (observed by their respective sensors), possibly weighted by the outside and return damper positions if available. In our proposed portable programming model, an application developer only needs to refer to the mixed air temperature CQ (\( T_{ma} \)); the definition of the CQ handles the complexity of determining which data sources to use in delivering the mixed air temperature to the application when it is executed.

4.2.2 Formal Definition. Formally, a CQ is a list of \( n \) functions (meaning \( n \) different ways to deliver the quantity), where each function takes a metadata graph \( G \) as an argument and returns either a vector of real numbers (\( \mathbb{R}^d \)) or a null value (\( \emptyset \)):

\[
CQ_i := (f_1, f_2, \ldots, f_n),
\]

where \( f_j : G \rightarrow (\mathbb{R}^d, \emptyset) \forall j \in 0, \ldots, n \) (3)

Each function \( f_j \) represents a possible implementation of the CQ; executing this function retrieves data corresponding to the logical value represented by the CQ. A CQ’s defining functions are ordered by how accurate or otherwise how desirable the implementations are. In our example of the \( T_{ma} \) CQ, finding a mixed air temperature sensor would be more desirable than estimating mixed air temperature from damper positions and upstream temperature sensors since it observes the quantity directly. Therefore, the function representing an implementation with a mixed air temperature sensor would be higher in order than the function performing an estimation.

The viability of an implementation must be determined with respect to a given metadata graph \( G \). Specifically, an implementation is viable if both the metadata requirements and the data requirements (specified by the application – see §4.4) can be satisfied by the graph content. The process of finding a viable implementation of a CQ is called resolution and it is discussed in §4.4.

The two viability properties can be determined in different ways. We have developed three types of CQ implementations that capture common and expressive strategies for discovering data in a portable manner: Graph CQs, Virtual CQs, and Default CQs.

4.3 CQ Implementations

A CQ implementation can be viable on multiple parts of a metadata graph. In our running \( T_{ma} \) example, if there are multiple AHUs in a building, then the portable programming model should attempt to find a viable implementation for each of them.

4.3.1 Graph CQs. A Graph CQ is defined by a SHACL shape which encodes constraints on the graph. Graph CQs search the metadata graph \( G \) for data sources with the desired properties and semantics for the CQ, as encoded in the SHACL shape. These shapes must define exactly one target and exactly one point. A shape’s target specifies what kinds of entities in the graph should be subject to the shape’s constraints (e.g. “all AHUs in the building”). §4.4 explains...
We can now describe the portable programming model and the values and scientific constants.

A shape’s point specifies the CQ’s data source and how it should be related to the target. Consider defining a Graph CQ implementation for our running \( T_{ma} \) example. To encode an implementation that identifies a sensor directly observing the mixed air temperature region of an AHU, one would create a SHACL shape with a target of the brick:Air_Handling_Unit class and a point of the brick:Mixed_Air_Temperature_Sensor class. See Figure 2 for the full SHACL shape definition. A shape can also encode other constraints that ensure that the target is appropriate. In Figure 2, we might add an additional clause that only matches single zone air handling units with no downstream terminal units.

### 4.3.2 Virtual CQs

A Virtual CQ is defined by a scalar function over other CQs. This allows CQs to be constructed from other CQs. Like all CQs, a Virtual CQ takes a metadata graph \( G \) as an argument; it passes the graph to all of its constituent CQs when required during the resolution process (§4.4).

Consider defining a Virtual CQ implementation for our running \( T_{ma} \) example. To encode an implementation which estimates the mixed air temperature from the outside and return air temperature sensors and corresponding damper positions, one could compute the following expression inside a Virtual CQ:

\[
T_{ma} = T_{ra} + D_{ra} + T_{oa} + D_{oa}
\]

Above, \( T_{[ma,oa]} \) represents CQs for the temperature of mixed, return and outside air; and \( D_{[ra,oa]} \) represents the position of the return and outside air dampers.

Virtual CQs allow developers to reuse other CQs in their own applications. By giving developers the means to construct their own representations of building data, we can help abstract away the complexity of data discovery.

### 4.3.3 Default CQs

A Default CQ is a constant expression that returns a single (possibly multi-dimensional) value. This serves two purposes. First, it can act as a fallback definition for CQs that might be hard to calculate. In our running \( T_{ma} \) example, we use a common estimation of 1.1 °C for the temperature drop across the supply fan [31] as a backup value for when direct sensing of that quantity is not possible. Second, Default CQs can act as aliases for common values and scientific constants.

### 4.4 CQ Resolution

We can now describe the portable programming model and the resolve method central to its implementation. The resolve method allows the developer-facing implementation of portable applications to remain straightforward and mostly about the application logic itself. It does this by finding the implementations of each CQ in an application that are most appropriate for a given metadata graph. The output of the resolve method is a set of functions; each function corresponds to a specialized implementation of the original application for a particular part of the building.

Formally, we can represent a portable application \( A \) as a function over one or more CQs. Below, \( T \) represents a desired temporal configuration, which encodes requirements on the necessary extent, sample rate, and aggregation of the CQ-defined data:

\[
A^T : f(CQ_1, CQ_2, \ldots, CQ_n) \rightarrow \text{user-defined output}
\]

Res\(\text{olve}\) is a function which takes an portable application \( A^T \), a metadata graph \( G \) and a timeseries database \( DB \) as arguments and returns a set of ported functions. A ported function is a copy of the application where each CQ has been bound to actual values corresponding to the temporal specification \( T \) for a particular target in the building. Resolve returns multiple functions because an application may be able to run in multiple contexts. In our running example of an AHU fault detection rule, the application \( A \) would represent the generic (i.e., portable) logic of checking the fault condition. If multiple AHUs exist in a building, then res\(\text{olve}\) would return a function for each AHU. Each function is a copy of the original application but with the CQs substituted for the actual sensor values for a specific AHU.

Formally, we define res\(\text{olve}\) as the function:

\[
\text{resolve} : (A^T, G, DB) \rightarrow \{A^T_i | \forall t \in \text{Targ}[i] | Vi \in \text{CQ}[A^T_i]\}
\]

Targ\([i]\) returns the set of entities in the graph \( G \) where a CQ\(i\) has a viable implementation. CQ\(A^T_i\) returns the set of CQs used within \( A \). The expression in Equation 6 above finds the set of targets \( t \) which have a viable implementation for each CQ \( A^T_i \) by the application \( A \). This means finding data for each logical quantity in the application is possible. \( A^T_i \) represents a transformed copy of \( A^T \) in which each CQ has been substituted with the corresponding data value found by an implementation of that CQ.

Res\(\text{olve}\) works by finding the best implementation of each CQ for each target \( t \) of the application. The “best” implementation of a CQ is the one that (a) appears earliest (left-most) in its defining list (Equation 3), and (b) points to data in the timeseries source \( DB \) that fulfills the application’s temporal specification \( T \). A ported function is a copy of \( A^T \) with each constituent CQ replaced by one of its implementation functions. Res\(\text{olve}\) ensures that each implementation relates to the same target, i.e. entity within the building model.

Figure 3 illustrates a single application specification being transformed into multiple functions which implement the application logic. Their execution produces the application results. Section 5 illustrates an example of res\(\text{olve}\) porting our FDD example 1.

### 4.5 Addressing Portability Challenges

Computational Quantities and the res\(\text{olve}\) method allow applications to be defined in a portable manner. We detail here how our proposed approach handles each major portability challenge:

#### 4.5.1 Model Expressivity (C1)

There are two ways that our proposed approach handles model expressivity. First, graph CQs embed SHACL shapes that are expressive enough to capture multiple ways
Temporal requirements exist on the application; CQs only handle developers straightforwardly express application logic using CQs asnant data sources for applications, but only when those data sourcesvariance within a metadata model.

must be constructed with an awareness of the portability challenges placeholders for building data and common quantities. While CQs burden of configuring an application to the definition of CQs. De-
to the application developer. We accomplish this by shifting the
and deploying them across many heterogeneous buildings with lit-
accuracy of a CQ's implementation and the availability of data.

Future work will examine more nuanced tradeoffs between the
temporal requirements (e.g. temporal extent, sampling rate) of the
resolve

CQs, interpreted through the resolve method, provide applications with the abstraction of having exactly the data they need.

4.5.3 System Applicability (C3). The SHACL shapes embedded inside graph CQs not only define the relationship between the target and the data source, but can also capture additional requirements about either the target or data source. We get this feature for “free” because we use the SHACL standard directly.

4.5.4 Temporal Configuration (C4). Many building applications define temporal requirements for the data they use. For example, ASHRAE’s Guideline 36 specification of high-performance sequences of operation requires all data sources for fault detection rules to use “five-minute rolling averages with 1-minute sampling time”[19]. Our proposed portability mechanism captures the temporal requirements and passes them to the resolve function. Temporal requirements exist on the application; CQs only handle how data can be found in the model. resolve incorporates the temporal requirements (e.g. temporal extent, sampling rate) of the application when determining the suitability of a CQ implementation. Future work will examine more nuanced tradeoffs between the accuracy of a CQ’s implementation and the availability of data.

5 LIFECYCLE OF A PORTABLE APPLICATION

Our portable programming model enables writing applications once and deploying them across many heterogeneous buildings with lit-
tle or no manual reconfiguration and with minimal complexity to the application developer. We accomplish this by shifting the burden of configuring an application to the definition of CQs. Developers straightforwardly express application logic using CQs as placeholders for building data and common quantities. While CQs must be constructed with an awareness of the portability challenges

5.1 CQ Specification

Recall that a CQ is formally defined (section 4.2.2) as a list of possible implementations of the CQ inside a graph model (i.e. building) listed from the most to the least desirable. In other words, a CQ specification encapsulates all the possible ways of discovering a specific quantity within a building, enabling its self-configuration in the next steps.

Figure 4 illustrates how applications can be constructed from an existing library of CQs. CQs themselves can have multiple possible implementations: the bottom of Figure 4 illustrates the four possible implementations of the \( T_{ma} \), ordered from most to least preferable. CQs are defined in Python; the entirety of the \( T_{ma} \) definition is given by Figure 5. GraphCQs can be defined either by (1) referring to an externally-defined shape (line 9), or (2) inline (lines 11-15) using shorthand for common SHACL constructions. The GraphCQ on lines 11-15 also captures contextual information about the AHU (requiring a mixed air damper) that must be true for that implementation to be considered.

Figure 2 displays the definition of the shape referred to on line 9 of Figure 5. Referring to externally-defined shapes eases integration with emerging metadata authoring frameworks like [18].

5.2 Authoring Portable Applications using CQs

We can now address how developers use CQs to express portable building applications. Recall from the definition in Equation 5 that an application is a function over a set of CQs. In our Python-based framework, the CQs used by an application must be expressed as
1 from SeeQ import SeeQ
2 from APAR.CQs import Toa, Tra, Fra, Foa
3
4 AHU_Tma = CQ(description="Mixed Air Temperature in the AHU",
5 unit=UNIT.DEG_C, 
6 implementations=[
7 # defined by the shape in Figure 2
8 GraphCQ(URIRef("urn:Tma_direct_observation")),
9 # using shorthand to define a SHACL shape
10 GraphCQ{
11 [BRICK.AHU],
12 [BRICK.hasPoint],
13 [BRICK.Mixed_Air_Temp_Sensor]
14]
15 VirtualCQ(Tra*0.5+Toa*0.5)
16])

import SeeQ

Figure 5: Example CQ Tma specification. CQs are specified in the back-end and can be reused by application developers.

Figure 6: Expressing the application logic of our two running examples using a set of pre-specified CQs

5.3 Resolving and Executing Applications

Running a portable application on a new deployment site requires two steps: (1) resolving it over a graph model to produce a building-specific implementation which is then (2) executed to perform computation and generate results. The resolve function (§4.4) takes as input an application A, a metadata graph G representing a building, and a source of timeseries data DB. Resolving the application produces a list of ported functions each corresponding to an execution of the application on some entity in the model (Figure 3).

Figure 7 shows how to port an application in our Python framework’s resolve function along with the Brick model of the building (in the graph object g) and a timeseries database client. The resulting value, APAR_R1_impls, is a list of functions in which the function parameters have been bound to dataframes of timeseries data pulled from the database. We annotate each ported function with the name of the entity it corresponds to (the target attribute). Executing the ported function delivers the results for that entity.

5.4 Brick Model Creation

Creating metadata models that support portable applications can be a challenge due to the complexity of buildings and the resulting complexity of the semantic metadata standards developed to represent them. BuildingMOTIF [18] is a recent tool for using SHACL-based descriptions of application metadata requirements (“manifests”) to drive the creation and validation of building metadata models. This process is guided by the principle of semantic sufficiency, which states that a building model is “complete” when it contains sufficient metadata to support a specified suite of applications.

Because our CQ portability mechanism is built on SHACL, we can use the set of CQs in an application to automatically create the SHACL-based manifest required by BuildingMOTIF. This is significant because application developers do not need to go through the trouble of defining the metadata requirements for their application – these can be derived automatically from the application’s definition. This also allows users of portable applications to test whether or not an application will run on a model, as well as check what metadata might be missing from their model.

6 EVALUATION

We evaluate the proposed programming model in three ways: (1) exercising the ability of SeeQ to author and run five categories of data-driven building applications, (2) testing self-configuration across diverse deployment sites, where existing approaches would otherwise fail, (3) comparing the development effort required among existing application portability mechanisms (i.e., Mortar, Energon).

6.1 Demonstration of Use

To evaluate the efficacy of the proposed abstraction, we have authored, resolved and executed 5 diverse data-driven applications across 76,100 sq. meters (Figure 8).
as (4) leaking valves. We authored the application using 8 CQs, with reheating capability when control sequences are not running.

Section 5.6.6 Appliance Rule Set Description. This section describes a set of alarms raised in VAV units.

Figure 9: Distribution histogram of how APAR CQs are used to express the entire APAR set of 28 rules.

6.1.3 KPI-driven Occupancy Density. Understanding occupancy density over the opening hours of a building into three groups and a specific period of time. We categorise the average occupant density over the opening hours of a building into three groups and estimate it using three CQs: the opening, and closing time of the building and the maximum capacity of each room (i.e., implemented as DefaultCQs), and a CQ representing the number of occupants.

over a specific period of time. We categorise the average occupant density over the opening hours of a building into three groups and estimate it using three CQs: the opening, and closing time of the building and the maximum capacity of each room (i.e., implemented as DefaultCQs), and a CQ representing the number of occupants.

Figure 10: Fault Detection using APAR R1. Faulty Operation detected during heating mode (and highlighted in red).

6.1.1 APAR for FDD on AHUs. APAR is a fault detection tool that uses a set of 28 expert rules, relying only upon sensor data and control signals that are commonly available in commercial automation and control systems. We specify and express the logic of the entire APAR rule set, defining and reusing 21 different CQs. Figure 9 displays the distribution of CQs across 28 APAR rules, demonstrating their ability to be reused across the entire rule set, thus reducing repetitive efforts during application authoring. Resolving and executing APAR Rules across AHUs can identify periods of abnormal operation: APAR R1 is being violated in Figure 10.

6.1.2 G36 Sequences for VAVs with Reheat. ASHRAE’s G36 specification provides a set of uniform sequences of operation intended to provide control stability of air-side HVAC systems. Here, we adopt G36’s Section 5.6.6 to describe a set of alarms raised in VAV units with reheating capability when control sequences are not running properly. Including our running example 2 of leaking damper, this application suite is also able to detect (1) low airflow, (2) low discharge air temperature, (3) poorly calibrated air flow sensors as well as (4) leaking valves. We authored the application using 8 CQs, and resolved it over our 5 deployment sites to produce executable implementations for 36 VAV units with reheating coils.

6.1.3 KPI-driven Occupancy Density. Understanding occupancy patterns in a group of buildings is an essential task, supporting from simple maintenance scheduling tasks to more advanced operational tuning of building systems. Here, we employ SeeQ to author a portable KPI-driven application for measuring occupancy density over a specific period of time. We categorise the average occupant density over the opening hours of a building into three groups and estimate it using three CQs: the opening, and closing time of the building and the maximum capacity of each room (i.e., implemented as DefaultCQs), and a CQ representing the number of occupants. We express this application in eq.7, author it using only few lines of code in Figure 11, resolve it across our 5 Brick models and execute it in 152 applicable zones.

$$\theta_d = \sum_{k=\text{open}}^{\text{closed}} \frac{\sigma_{\text{occ}, k}}{\sigma_{\text{occ}, \text{max}} \cdot \sum_k}$$

6.1.4 ZonePAC for Virtual Sensing. To evaluate SeeQ on portable virtual sensing applications, we developed a partial portable implementation of ZonePAC [8, 22], a HVAC metering application that employs a heat transfer equation and a linear regression model to estimate the cooling thermal power of VAV boxes. In Figure 12, we are quoting two portability challenges faced by Koh et al., while trying to implement this application at scale. CQs allow users to incorporate alternative implementations, that can then enable the resolution of the application in many possible ways.

6.1.5 Random Forest for Zone Air Temperature Prediction. Despite the growing advances of Machine Learning (ML) applications, it remains challenging to rapidly deploy ML models in a group of buildings. In Figure 13, we demonstrate how SeeQ can interface with a popular ML library (e.g. scikit-learn) to author and implement a portable version of a Random Forest algorithm for the prediction of zone air temperature, enabling users to reuse CQs as features for the development of ML models.
Example of a Python code snippet:
```
from pandas import DataFrame
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

def linear_regression(sat, fan, coil):
    X = assemble_dataset(sat, fan, coil)
    y = price
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    model = LinearRegression()
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)
    return model and test MSE
```

Table 1: Implementation Description of Portable APAR

<table>
<thead>
<tr>
<th>AHU</th>
<th>Implementation[CQ_type#no_impl]</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHU07</td>
<td>Tma: GraphCQ (#1)</td>
</tr>
<tr>
<td></td>
<td>(bldgB) GraphCQ (#1)</td>
</tr>
<tr>
<td>AHU02</td>
<td>Tma: VirtualCQ (#3) = Tra&quot;Dra+Toa'Doa)</td>
</tr>
<tr>
<td></td>
<td>(bldgA) GraphCQ (#2)</td>
</tr>
<tr>
<td>AHU15</td>
<td>Tma: VirtualCQ(#4)=Tra&quot;0.5+Toa&quot;0.5</td>
</tr>
<tr>
<td></td>
<td>(bldgD) Tsa: VirtualCQ (#2) = Tdp+(Rt-Dtp)/3</td>
</tr>
<tr>
<td></td>
<td>DefTsf: DefaultCQ(#1)</td>
</tr>
<tr>
<td></td>
<td>Epsilon, t: DefaultCQ(#1)</td>
</tr>
<tr>
<td></td>
<td>Hhc: GraphCQ(#1)</td>
</tr>
</tbody>
</table>

Table 2: Lines of Code required: Mortar vs Energon vs SeeQ. Table updated from [21]

<table>
<thead>
<tr>
<th>Application</th>
<th>Mortar</th>
<th>Energon</th>
<th>SeeQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building Integrated Control</td>
<td>42</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>Energy Consumption Prediction</td>
<td>54</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>AHU Fault Detection</td>
<td>73</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>Chiller Profiling</td>
<td>52</td>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 14: Part of a SPARQL Query attempting to extract Tma and Tsa for our Running Example of APAR R1

6.3 Development Effort

SeeQ provides a framework for expressing application requirements and logic, without considering a specific deployment site. Instead, existing approaches like Mortar [17] and Energon [21] facilitate the retrieval of building metadata, leaving users to configure applications across different buildings. To achieve that, we encapsulate the burden of configuration inside CQs. This abstraction layer allows developers to express their application logic, by reusing CQs (Figure 9). Thus, unlike in prior systems, they do not need to write any queries or handle the portability of applications themselves.

As illustrated throughout this section, our programming model can retrieve relevant metadata and express applications in 2 lines of code; this might take from 50-100 lines of code in Mortar [17] to 5-10 lines of code in Energon [21]. In Table 2, we compare how SeeQ further reduces costs of metadata retrieval compared to Mortar and Energon, while, in addition, incorporates applications’ requirements and logic. Hence, SeeQ not only reduces the effort to retrieve metadata, omitting the familiarity with querying language and graph structures, but also allows developers to author portable applications with a few lines of code.

Developing portable building applications using Mortar [17] and Energon [21] depends on the ability of developers to write successful queries, using SPARQL or EnergonQL, respectively. However, writing a query to capture various ways of discovering the required application metadata can be cumbersome, ineffective and exponentially complex as the number of application variables increases. We demonstrate this complexity through Figure 14 that displays a part of a SPARQL Query, written to retrieve Tma and Tsa from our APAR R1 running example. Capturing the various possible implementations for only two data points, required multiple and complex "UNION" statements, demonstrating the inability of this approach to scale. This example underlines the potential of SeeQ to address portability challenges and simplify application development.
7 CONCLUSIONS
In this paper we have introduced SeeQ, a programming model for portable building applications built on semantic metadata models like Brick. The programming model is built on a proposed abstraction of computational quantities (CQs), which are reusable self-configuring identifiers. CQs reduce and simplify development effort by abstracting away the difficulty of finding data in heterogeneous buildings. The resulting applications are orders of magnitude smaller than in prior programming models while also providing portable execution.

This work marks a significant step towards turnkey applications for building stakeholders. Reducing development and deployment costs through portable applications helps to democratize adoption of state-of-the-art analytics including occupant-comfort and energy-saving approaches.

ACKNOWLEDGMENTS
This research has been co-funded by the European Union’s Horizon research and innovation programmes; CBIM-ETN under Marie Sklodowska-Curie grant agreement No 860555, and DigiBUILD under agreement No 101069658.

REFERENCES