Mortar: An Open Testbed for Portable Building Analytics

GABE FIERRO, UC Berkeley
MARCO PRITONI, Lawrence Berkeley National Laboratory
MOUSTAFA ABDELBAKY, DANIEL LENGYEL, and JOHN LEYDEN, UC Berkeley
ANAND PRAKASH and PRANAV GUPTA, Lawrence Berkeley National Laboratory
PAUL RAFTERY, THERESE PEFFER, GREG THOMSON, and DAVID E. CULLER,
UC Berkeley

Access to large amounts of real-world data has long been a barrier to the development and evaluation of analytics applications for the built environment. Open datasets exist, but they are limited in their span (how much data is available) and context (what kind of data is available and how it is described). Evaluation of such analytics is also limited by how the analytics themselves are implemented, often using hard-coded names of building components, points and locations, or unique input data formats.

To advance the methodology for how such analytics are implemented and evaluated, we present Mortar: an open testbed for portable building analytics, currently spanning 90 buildings and containing over 9.1 billion data points. All buildings in the testbed are described using Brick, a recently developed metadata schema, providing rich functional descriptions of building assets and subsystems. We also propose a simple architecture for writing portable analytics applications that are robust to the diversity of buildings and can configure themselves based on context. We demonstrate the utility of Mortar by implementing 11 applications from the literature.

CCS Concepts: • **Information systems** → *Graph-based database models*; *Information retrieval*; • **Computing methodologies** → *Model development and analysis*;

Additional Key Words and Phrases: Smart buildings, modeling and analytics, dataset

ACM Reference format:
Gabe Fierro, Marco Pritoni, Moustafa AbdelBaky, Daniel Lengyel, John Leyden, Anand Prakash, Pranav Gupta, Paul Raftery, Therese Peffer, Greg Thomson, and David E. Culler. 2019. Mortar: An Open Testbed for Portable Building Analytics. *ACM Trans. Sen. Netw.* 16, 1, Article 7 (November 2019), 31 pages.
https://doi.org/10.1145/3366375

This research is supported in part by California Energy Commission EPC-15-057, Department of Energy grant EE-0007685, and the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.
Authors' addresses: G. Fierro, M. AbdelBaky, D. Lengyel, J. Leyden, G. Thomson, and D. E. Culler, UC Berkeley, 410 Soda Hall #1776, Berkeley, California, 94720-1776, USA; emails: {gtfierro, moustafa}@cs.berkeley.edu, {daniel.lengyel, jleyden, thomsong}@berkeley.edu, culler@cs.berkeley.edu; M. Pritoni, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Office 090-2066, Berkeley, California, 94720, USA; email: mpritoni@lbl.gov; A. Prakash and P. Gupta, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Office 090-2056E, Berkeley, California, 94720, USA; emails: {akprakash, pranavhgupta}@lbl.gov; P. Raftery, UC Berkeley, 390 Wurster Hall #1839, Berkeley, California, 94720-1839, USA; email: p.raftery@berkeley.edu; T. Peffer, UC Berkeley, 2150 Allston Way, Suite 280, Berkeley, California, 94704-1346, USA; email: tpeffer@berkeley.edu.

1550-4859/2019/11-ART7 $15.00
https://doi.org/10.1145/3366375

1 INTRODUCTION

Building analytics applications vary in complexity from simple algorithms that only require a few directly related data streams to algorithms using complex white- or gray-box models, requiring detailed information about the building. This information typically must be acquired from diverse sources, such as from a building automation system, architectural and mechanical drawings, operations and maintenance documents, and human input from the building operator. Implementations of analytics applications are often done as "one-offs" because of the high degree of site-specific application logic. For example, this logic can be based on the types of subsystems in the building, the names of BMS points needed for the application, and how those BMS points relate to spatial elements of the building. As a result, many analytics algorithms go largely untested and unevaluated beyond the handful of buildings (or simulations) the algorithm was developed against.

This pattern produces implementations that suffer from bias and have little generalizability [19]. A recent evaluation of U.S. building energy benchmarking and transparency programs [27] found that "indications of [energy] savings should be considered preliminary... because of the limited period of analyses and inconsistencies among analysis methods for the various studies." This is a lost opportunity for robust evaluations of analytical applications.

The existence of a large collection of building data would grant data scientists, algorithm developers, and building managers the opportunity to empirically evaluate their work on a more diverse body of buildings. Standard, open datasets and workloads exist in many other areas (such as TPC-C [37] for relational databases and MNIST [23] for digit recognition). There are several existing and prior efforts working to provide such an open dataset for buildings [6, 7, 13, 21, 26, 40]. However, these datasets are limited in their span (how much data is available) and context (what kind of data is available and how it is described), which limits their efficacy for the evaluation of analytics applications.

In addition to the need for comprehensive, up-to-date, and well-annotated datasets, we also need to make applications portable in order to minimize the effort in running analytics applications against different buildings. Portable applications can nimbly provide retrocommissioning, fault detection, and diagnostics across a wide range of buildings with minimal reconfiguration.

The goal of this work is to address these challenges by providing an open testbed and a platform for storing, describing, updating, discovering, and retrieving building data. The testbed currently spans 90 buildings and contains over 9.1 billion data points. We utilize and extend Brick [5], a recently developed metadata schema that provides rich functional descriptions of building assets and subsystems, to describe not only data streams but the mechanical, electrical, logical, and functional context of those streams within all buildings in the testbed. Using this standardized and extensible method of describing buildings, we show that it is possible to create a data platform that can subsume the heterogeneous data produced by the built environment. Further, we also present an architecture for portable and extensible analytics applications that leverages Brick to simplify the development and evaluation of these applications across multiple buildings.

Together, the testbed and the portable architecture provide a platform, named Mortar, which lowers the integration cost of deploying new analytics applications and acts as a vehicle for reproducible evaluations. We evaluate Mortar by implementing a representative set of analytics applications. Further, we release these implementations as well as the datasets and Brick models that are part of the testbed for others to leverage and build upon.

The Mortar platform hosts the metadata and timeseries data for a growing collection of buildings. It provides access to that data for a repository of open-source, vetted, and robust implementations of building analytics algorithms.[1] We intend for this platform to be used

[1]Available online at https://github.com/SoftwareDefinedBuildings/mortar-analytics.

by algorithm developers and researchers for the implementation and evaluation of analytics applications, and by building managers who upload and describe their building data in order to run any available analytics to manage their portfolio. The timeseries data, Brick models, and analytics library mentioned in this article are freely available as part of the Mortar platform (available online at https://mortardata.org/ with creation of a free account); the platform will soon support user-provided datasets and metadata models.

The contributions of this article are as follows:

(1) The design and implementation of a platform for the development and evaluation of portable building analytics.
(2) A modular and extensible architecture for portable analytics applications.
(3) A dataset of 90 buildings containing over 9.1 billion data points, where the relationships between data points are well described and annotated using Brick models

This article is an extension of a previously published work [15] with the following additions:

(1) A discussion of the design of the Mortar query processor implementation (Section 4), which has been deployed in production serving over 25,000 dataset requests in the last few months.
(2) The design and implementation of Mortar's declarative interface for describing large multi-dimensional datasets (Section 4).
(3) A set of case studies detailing the experience and results of implementing six applications using Mortar (Section 8).

The remainder of the article is organized as follows. Section 2 provides a brief background on Brick. Section 3 and Section 4 detail the design and implementation of Mortar. Section 5 presents the architecture of portable applications. A detailed description of the building datasets is presented in Section 6, followed by the evaluation of Mortar in Section 7 and the application case studies in Section 8. Related work is presented in Section 9 and the article concludes in Section 10 outlining future work.

2 BRICK

Brick [5] is an open-source ontology providing a unified semantic representation of building assets, subsystems, and the relationships between them. Brick has two components: an extensible *class hierarchy* representing the physical and logical entities in buildings, and a minimal set of *relationships* that capture the connections between entities. A Brick model of a building is a labeled, directed graph in which the nodes are entities and the edges are relationships.

2.1 Why Brick?

Brick has several advantages over alternative metadata representations such as Project Haystack [1] and IFC [8].

(1) Completeness: Brick's class hierarchy can describe the diverse array of equipment, points, subsystems, and other physical or logical assets found in buildings. The class hierarchy can be extended to define new or unusual entities.
(2) Flexibility: by utilizing the class hierarchy and transitive relationships in queries against Brick models, applications can remain agnostic to the level of detail of a Brick model. This allows applications to run on buildings whose Brick model may not be fully specified, but contains the points and relationships necessary for the application to run.

Table 1. Proposed Set of Brick Extensions for Capturing Building Properties

Property	Definition
`hasSite/isSiteOf`	`brick:Site` entity to capture points belonging to a single facility.
`noaa`	Nearest NOAA weather stations. Gives rough location of site.
`area`	Floor area (typically m^2 or ft^2). Can be associated with `brick:Site` (for total area) or `brick:Floor`, `brick:Room`, and so forth, for more fine-grained annotation
`adjacentTo`	Adjacency for rooms, other spaces.
`uuid`	Timeseries identifier used by the timeseries database.
`hasUnit`	Engineering units for the associated timeseries.

(3) Consistency: the minimal set of relationships defined by Brick and the constraints defined by the Brick ontology help reduce variability in how the same concepts are expressed across independent Brick models.

(4) Queryability: applications use the powerful SPARQL [18] language to query Brick models. SPARQL enables applications to traverse the complex and heterogeneous graph structures that typify Brick models and retrieve the information they need to operate. [18] provides a full explanation of the SPARQL language and [14] contains a discussion of the application of SPARQL to Brick.

These properties make Brick an ideal candidate for addressing the metadata-related shortcomings in existing platforms. Further, Brick lays the foundation for developing and evaluating portable analytics applications, which is described in detail in Section 5.

2.2 Brick Extensions

Although the current release of the Brick schema (1.0.3) can describe all of the equipment and points contained in the testbed, it does not contain site-level properties such as building typology that are helpful for qualifying which sites to include in an analysis. Table 1 lists the properties we have added to Brick; these changes are being collected into a formal proposal for inclusion into an upcoming release of Brick.

3 MORTAR SYSTEM ARCHITECTURE

Mortar stands for **M**odular **O**pen **R**eproducible **T**estbed for **A**nalysis & **R**esearch. It provides a platform for the development and evaluation of portable building analytics applications. Mortar contains timeseries data consisting of historical values of the sensors, actuators, setpoints, and other data points for a large number of buildings. Mortar manages a Brick model for each of these buildings, which describes the properties of the building as well as the equipment, sensors, subsystems, and relationships within that building. Mortar also includes a library of analytics applications, structured according to the architecture detailed in Section 5. These applications access timeseries data by executing queries against the Brick model to refer to relevant collections of data streams. This methodology enables portable analytics applications, made more robust through evaluation against the diverse population of buildings in the testbed.

The architecture of Mortar is driven by the need to (a) link historical timeseries data with its context as captured by a Brick model, (b) execute code that retrieves segments of timeseries data filtered by context, and (c) provide those features at the scale of tens or potentially hundreds of buildings. Mortar is assembled from a family of open-source software. Mortar has four main components: a timeseries storage, a Brick model storage, a query processor, and an application execution environment (as shown in Figure 1).

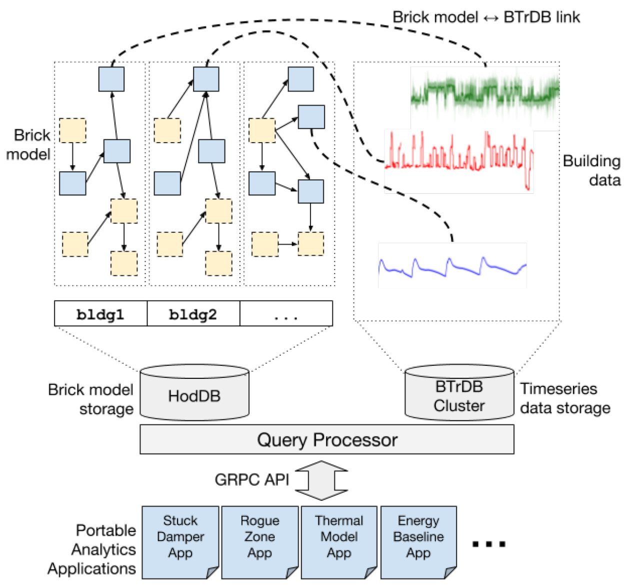


Fig. 1. Mortar architecture.

Timeseries Storage. The testbed manages the distribution and updating of data using a full timeseries database instead of distributing data in static file form (such as CSVs). At the testbed's current scale of over 145 GB of 64-bit floating-point values and 64-bit nanosecond-precision timestamps, it is unreasonable to expect users to download the data themselves. Many timeseries databases offer filtering and aggregation features that allow applications to specify and download just the data they need for their execution.

The process of appending new data to an existing stream is also made much easier through the use of a timeseries database. One of the critical features of the testbed is its support for progressively updated datasets ("live" data). Building managers and other users need to be able to upload recent data for existing building points as well as upload historical data for newly annotated building points.

Brick Model Storage. Another key aspect of the architecture is the storage of Brick models over time, and the integration of the Brick model with the timeseries database. Mortar needs to store a Brick model for each building and maintain the history of those models as they are changed and updated. Brick models contain entities called *Points* which represent measurable quantities in a building such as sensors, setpoints, alarms, and equipment statuses. Points are sampled and stored in a timeseries database under some unique identifier; the timeseries database uses 36-byte UUIDs. These identifiers are stored in the Brick model as string literals and are associated with Points via the `brickframe:uuid` property. In this way, Brick model decouples the context and name of a point from its timeseries representation, which opens the possibility for the testbed to change the timeseries representation without affecting how applications are run. This is helpful for maintaining continuity of data collection through repairs and retrofits, processing data (e.g., for re-uploading data with an adjusted calibration parameter), and in cases of fixing errors (e.g., if the wrong data was uploaded for a building point). Because the qualification and execution of applications makes heavy use of Brick queries, it is important that the storage medium for the Brick model provides low-latency query response times despite a large number of concurrent requests against a large number of buildings.

Query Processor. Applications interact with the Mortar platform via the query processor's API, which provides three methods.

(1) `GetAPIKey(username, password)` returns a temporary token that can be used by applications to authenticate to the Mortar API under a user's account.
(2) `Qualify(list of Brick queries)` returns a list of Mortar sites that return results for each of the provided queries. This simplifies the process of identifying for which sites a Mortar application can run, and will be increasingly helpful as the size of the Mortar dataset grows.
(3) `Fetch(dataset description)` takes as an argument a declarative specification of the metadata and timeseries data required by an application and streams the resulting dataset to the user, where it can be constructed into a representation of the client's choice. The streaming mechanism and its interaction with clients is discussed in Section 4.

Application Execution Environment. Mortar applications are designed to run on compute resources external to the Mortar backend cluster. Applications are built on libraries such as PyMortar[2] which leverage Mortar's API to retrieve the data they need to execute. PyMortar can be installed through the PyPi[3] Python package repository or used as part of a virtualized workspace we have released as a Docker image.

The PyMortar Docker image contains a recent copy of the Mortar analytics library, an up-to-date version of the PyMortar library, and a Jupyter Notebook browser interface for programmatic interactions with the Mortar platform. Virtualizing the Mortar workspace in this matter facilitates deployment across platforms of varying capabilities: a building analyst can begin an experiment on their laptop with a small amount of data and then execute a larger version of the experiment on a capable server or cloud instance without spending significant time reconfiguring the programming and execution environment.

4 IMPLEMENTATION

The Mortar platform is live and available for use.[4] Our experiences with the live platform and workload have substantially influenced the design of the Mortar query processor since the initial release of Mortar in 2018 [15]. Here, we describe how the current implementation of Mortar addresses the following challenges:

— **Demand for Large Datasets:** data-driven applications such as regression-based energy baselining and thermal modeling can require years of high granularity historical data. Mortar's growing list of buildings and inclusion of live data mean that the amount of data these applications will require will only increase. Therefore, we have designed the Mortar backend to handle dataset requests of arbitrary size.
— **Growing User Base:** Mortar's growing user base means that the backend implementation needs to account for a potentially high number of concurrent, long-running queries.
— **Complex Application Logic:** prior releases of the Mortar platform only allowed users to define a single matrix of timeseries data with each request. This placed an undue burden on users to manage timeseries identifiers and contextual metadata between multiple requests for complex applications such as Virtual Coil Meter and Chilled Water Loop Total Electrical Consumption, which involve relationships between many different building assets. To

[2]https://mortardata.org/docs/pymortar/.
[3]https://pypi.org/project/pymortar/.
[4]https://mortardata.org/.

```
SELECT ?sensor ?setpoint ?equip ?sensor_uuid ?setpoint_uuid WHERE {
    ?sensor      rdf:type/rdfs:subClassOf* brick:Air_Flow_Sensor .
    ?setpoint    rdf:type/rdfs:subClassOf* brick:Air_Flow_Setpoint .
    ?sensor      bf:isPointOf    ?equip .
    ?setpoint    bf:isPointOf    ?equip .
    ?sensor      bf:uuid         ?sensor_uuid .     # AUTOGENERATED
    ?setpoint    bf:uuid         ?setpoint_uuid .   # AUTOGENERATED
};
```

Fig. 2. The metadata stage rewrites the View definition in Figure 3 to this query by inspecting the SELECT variables used in the request's DataFrame definitions. Here, those variables were ?sensor and ?setpoint.

simplify client-side logic, we have designed a new, expressive API for Mortar enabling users to retrieve multiple collections of metadata and timeseries data in the same request.

4.1 Declarative Dataset Description

The Mortar API Fetch method takes as an argument a declarative description of a dataset consisting of one or more tables of metadata called Views and any number of tables of timeseries data called DataFrames and returns that dataset.

A View definition consists of a name (unique to all other Views in the query), a Brick query, and a list of sites. Resolving a View returns a named table whose contents are the results of executing the Brick query against the Brick model for each of the listed sites. The table's columns are derived from the variables in the SELECT clause of the definition, with the addition of columns relating timeseries identifiers to variables (see Figure 2).

A DataFrame definition consists of a name (unique to all other DataFrames in the query), a list of timeseries, and an optional aggregation parameter with a bucket size. Resolving a DataFrame returns a table whose columns are timeseries identifiers and whose rows are the value of each timeseries at some UTC timestamp. Mortar supports several associative aggregation functions (min, mean, max, and count), but can also return the "raw" timeseries data from the database without any aggregation applied.

Each column of a DataFrame is a timeseries, labeled with its timeseries identifier (UUID). The set of timeseries in a particular DataFrame is given by a list of View columns whose values are Brick Points.

Consider the example Fetch request in Figure 3 for a Rogue Zone Detection application. The request defines one View that relates airflow sensors and setpoints to the equipment they are a part of—likely a Variable Air Volume Box. The application analysis deals with setpoint and sensor aggregates differently—max for setpoints and mean for sensors—and defines each collection in its own DataFrame. The ?sensor and ?setpoint variables in the definition of the View named airflow_points correspond to collections of Points that have timeseries data associated with them. The Mortar query processor expands these variables into the corresponding sets of UUID timeseries identifiers (using the brickframe:uuid property) when evaluating each DataFrame.

4.2 Backend Services

The Mortar backend consists of a family of services all managed by Kubernetes,[5] a container orchestration framework. Kubernetes simplifies several tasks:

- scaling the concurrency of the query processor by increasing the number of replicas (parallel, load-balanced copies of the query processor);
- scaling the amount of storage for Brick models and timeseries data by adding servers;

[5]https://kubernetes.io/.

```python
import pymortar

client = pymortar.Client()

request = pymortar.FetchRequest(
    # sites=[ <some list of sites here> ]
    views=[
        pymortar.View(
            name="airflow_points",
            definition="""SELECT ?sensor ?setpoint ?equip WHERE {
                ?sensor rdf:type/rdfs:subClassOf* brick:Air_Flow_Sensor .
                ?setpoint rdf:type/rdfs:subClassOf* brick:Air_Flow_Setpoint .
                ?sensor bf:isPointOf ?equip .
                ?setpoint bf:isPointOf ?equip
            };
            """
        )
    ],
    dataFrames=[
        pymortar.DataFrame(
            name="sensors",
            aggregation=pymortar.MEAN,
            window="30m",
            timeseries=[
                pymortar.Timeseries(
                    view="airflow_points",
                    dataVars=["?sensor"],
                )
            ]
        ),
        pymortar.DataFrame(
            name="setpoints",
            aggregation=pymortar.MAX,
            window="30m",
            timeseries=[
                pymortar.Timeseries(
                    view="airflow_points",
                    dataVars=["?setpoint"],
                )
            ]
        ),
    ],
    time=pymortar.TimeParams(
        start="2018-01-01T00:00:00Z",
        end="2019-01-01T00:00:00Z",
    )
)
response = client.fetch(request)
# => <pymortar.result.Result: views:1 dataframes:2 timeseries:835 vals:14628365>
```

Fig. 3. Declarative description of the `Views` and `DataFrames` required for the "Rogue Zone Airflow" application.

— deploying new versions of the Mortar backend with minimal disruption to existing traffic;
— continuous monitoring and crash recovery of backend processes.

Timeseries Database. The testbed uses the BTrDB timeseries database [4], which provides fast storage and retrieval of scalar-valued timeseries data. BTrDB supports adding storage capacity incrementally using the Ceph storage engine, allowing the testbed to scale gracefully as the number of buildings and amount of data in the testbed increases. BTrDB supports querying data at arbitrary resolutions, grouping data using associative statistical aggregates `min`, `mean`, `max`, and `count`, as well as querying the raw data itself. This reduces the processing that must be done by the query frontend.

BTrDB presents query results as a stream of timeseries data that must be consumed by a client. This reduces the amount of memory required for each executing query and helps BTrDB support many concurrent queries.

Brick Model Database. Existing open-source RDF/SPARQL databases do not perform well under a Brick model workload, often taking seconds or even minutes to execute a single query on a moderately sized Brick model [14]. Mortar stores all Brick models in HodDB [14], a Brick-specialized RDF/SPARQL database and query processor that demonstrates good performance on the testbed workload.

4.3 Query Processor

In order for the Mortar platform to handle the current and projected scale of requests, we have designed the Mortar query processor around a SEDA architecture. The SEDA architecture is characterized by distinct process stages connected by explicit queues [38]. Each SEDA stage has a number of parallel workers who take jobs from the stage's incoming queue, do some processing, and insert the job into the stage's outgoing queue. When a stage is overwhelmed–such as due to a lack of compute or memory resources—its incoming queue fills up. This means that the prior stage's workers will be unable to place their finished jobs in the incoming queue for the next stage, which influences the prior stage's incoming queue, and so on (this is called *backpressure*). A stage can react to backpressure in a few ways: it can add more workers to its stage to handle more requests at a time, or it may choose to do nothing or even reduce the number of workers, which can propagate backwards through each stage and reduce the load on the whole system. If the system is unable to cope with the incoming request load by adjusting internal resources, it can rate limit client requests.

After a prior version of the Mortar query processor crashed several times due to large data requests and high volumes of concurrent requests, we decided to adopt the SEDA architecture as a method of better managing server-side resources that could gracefully adapt to a varied, bursty workload. The query processor consists of three stages: **frontend**, **metadata**, and **timeseries**.

Frontend Stage. The frontend stage receives requests from clients in the form of calls to Mortar's GRPC API, prepares requests for execution, and delivers results back to the client using GRPC. The frontend authenticates each incoming request using the JWT (JSON Web Token) included in the request; clients can generate valid JWTs using the `GetAPIKey` API call. After authentication, the frontend stage checks that the request is valid and well-formed. The frontend stage then wraps the client's request in a *Context* object which encapsulates all resources allocated during query execution before attempting to place the request in the metadata stage's incoming queue.

Requests to Mortar's `GetAPIKey` API call are answered from the frontend stage alone. Requests to Mortar's `Qualify` API call flow through the frontend and metadata stages. Requests to Mortar's `Fetch` API call flow through all three stages. Rather than passing the full `Fetch` response between stages, the metadata and timeseries stages forward incremental parts of the response that can be reassembled by the client into the full dataset.

Metadata Stage. The metadata stage evaluates `Views` from incoming client requests by executing their Brick query definitions against a HodDB instance containing the Brick models for all sites in the Mortar testbed. `View` definitions do not include the binding between points and timeseries identifiers; instead, the metadata stage rewrites `View` definitions to query for the timeseries identifiers for points that are actually used in a `DataFrame`. For example, the `View` definition in Figure 3 would be rewritten to the query in Figure 2. This simplifies dataset definitions by making transparent the binding between an entity in the Brick model and its corresponding timeseries. To support this query rewriting functionality, we implemented an alternative query frontend to HodDB that accepts parameterized versions of SPARQL queries.

Timeseries Stage. The timeseries stage pulls data from BTrDB for each of the timeseries identifiers derived from the evaluation of the metadata stage's `Views`. The `TimeParams` field in the client request defines the temporal extent of the data to be retrieved, and the `DataFrame`

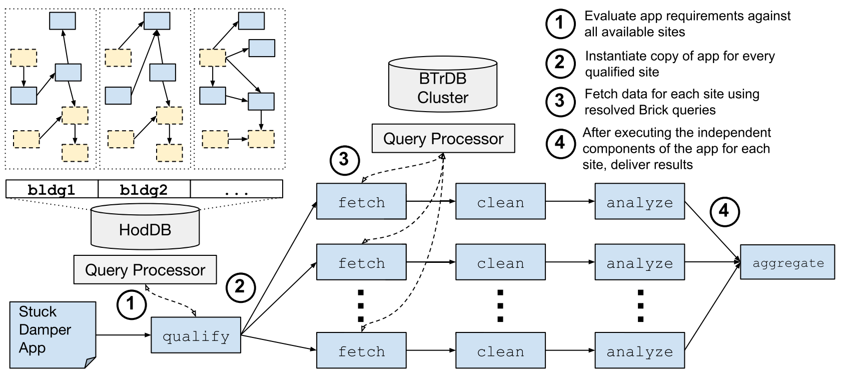

Evaluate app requirements against all available sites
Instantiate copy of app for every qualified site
Fetch data for each site using resolved Brick queries
After executing the independent components of the app for each site, deliver results
BTrDB Cluster
Query Processor
bldg1
bldg2
...
HodDB
Query Processor
Stuck Damper App
qualify
fetch
clean
analyze
aggregate

Fig. 4. Architecture of a portable analytics application. An application consists of five segments: `qualify` which filters the building corpus into the execution set; `fetch`, `clean`, and `analyze` which are all executed once for each building in the execution set; and `aggregate` which is executed once over all outputs of `analyze`.

definitions indicate which aggregation function to apply to each stream along with the desired resolution (see lines 43–45 of Figure 3).

Because a client can request tens or hundreds of megabytes of data, the timeseries stage takes care to minimize the performance impact of large queries or slow clients on the rest of the platform. To achieve this, the timeseries stage decomposes the requested timeseries data into small batches that are buffered in memory and enqueued for delivery to the client. The amount of buffered timeseries data is constant for all queries, so the amount of server memory dedicated to serving a request is independent of the size of the requested dataset. The timeseries stage only pulls data from BTrDB as needed, so if a client is slow to read incoming data or terminates the connection, the ongoing query to BTrDB can be terminated and the resources released without having read data unnecessarily.

5 ANATOMY OF PORTABLE APPLICATIONS

Portability measures how easily a program can be moved from one computing environment to another. In the context of building analytics, portability refers to how well a piece of code generalizes to multiple heterogeneous buildings. Many available implementations of building analytics are not portable due to hardcoded point names, assumptions about the structure of a building and its subsystems, assumptions about the availability of data, and tightly coupled phases of operation. These limitations are an inevitable consequence of how few real buildings are available for the development or evaluation of analytics applications, which makes it difficult to perform a robust evaluation.

In this section, we describe the general structure of a portable building analytics application designed to be executed against potentially hundreds of buildings with very little configuration. We decompose all Mortar applications into five components: `qualify`, `fetch`, `clean`, `analyze`, and `aggregate` (see Figure 4). These are executed in order by the platform when the application is invoked.

5.1 Application Requirements

The **`qualify` component** defines the metadata and data requirements of an application. Mortar evaluates these requirements against all available buildings in order to determine the subset

```
queries:
 required:
  - >
    SELECT ?vav ?sen ?sp WHERE {
      ?sp  rdf:type/rdfs:subClassOf* brick:Air_Flow_Setpoint .
      ?sen rdf:type/rdfs:subClassOf* brick:Air_Flow_Sensor .
      ?equip rdf:type/rdfs:subClassOf* brick:Terminal_Unit .
      ?sp  bf:isPointOf ?equip .
      ?sen bf:isPointOf ?equip .
      ?equip bf:feeds+ ?zone .
      ?zone rdf:type brick:HVAC_Zone .
    };
```

Fig. 5. YAML app specification for qualifying sites for “Rogue Zone (Airflow)” application.

of buildings against which the application can run (the *execution set*). The ratio between the execution set and the total number of buildings in the testbed provides a good measure of the portability of an application, and can be used iteratively, to allow an application to cover more buildings. Mortar forks an instance of the application for each site in the execution set. Specifically, the `qualify` component checks

(1) constraints on building typology and other properties, such as the number of floors in a building, floor area, climate, and occupancy class;
(2) data context constraints, such as the kinds of equipment in the building and available relationships; and
(3) data availability constraints, including the amount of historical data and available data resolution.

The first two constraints are defined using Brick queries. When qualifying an application, Mortar evaluates these queries against the Brick model for each site in the testbed; sites are placed in the execution set if the Brick queries return results. An application can use a set of Brick queries to implement a “decision tree” where the application customizes its execution based on the information available in the Brick model, such as to distinguish between RTU-based and Air Handling Unit (AHU)-based HVAC systems. Mortar evaluates data availability constraints against the data streams identified by the Brick queries.

An example `qualify` specification for the “Rogue Zone Detection”[6] application is shown in Figure 5. The Brick query looks for terminal units that expose air flow setpoints and sensors and relates the terminal unit to the HVAC zone it conditions. Brick’s subclass relationships allow the same query to be used to find systems with a single air flow setpoint and those with a high/low setpoint dead-band. In these cases where there is variability in the type of system or what data is exposed, it is up to the application developer how to deal with the difference: the developer can write more precise Brick queries to simplify the application code to only deal with a single type of system, or the developer can account for the potential differences thereby covering more buildings.

5.2 Data Retrieval

Mortar runs the **`fetch` component** for each site in the execution set. The `fetch` component performs the actual retrieval of data from the timeseries database corresponding to the set of streams identified by the Brick queries. The data retrieval request uses the following parameters:

[6]This application is described in more detail in Section 7.1.

```
def run(site):
  print 'running', site
  data = {}
  request = pymortar.FetchRequest(
    sites=[site],
    views=[
        pymortar.View(
            name="airflow_points",
            definition="""SELECT ?sensor ?setpoint ?equip WHERE {
                ?sensor rdf:type/rdfs:subClassOf* brick:Air_Flow_Sensor .
                ?setpoint rdf:type/rdfs:subClassOf* brick:Air_Flow_Setpoint .
                ?sensor bf:isPointOf ?equip .
                ?setpoint bf:isPointOf ?equip
            };"""
        )
    ],
    dataFrames=[
        pymortar.DataFrame(
            name="sensors",
            aggregation=pymortar.MEAN,
            window="10m",
            timeseries=[
                pymortar.Timeseries(
                    view="airflow_points",
                    dataVars=["?sensor"],
                )
            ]
        ),
        pymortar.DataFrame(
            name="setpoints",
            aggregation=pymortar.MAX,
            window="10m",
            timeseries=[
                pymortar.Timeseries(
                    view="airflow_points",
                    dataVars=["?setpoint"],
                )
            ]
        ),
    ],
    time=pymortar.TimeParams(
        start="2018-01-01T00:00:00Z",
        end="2019-01-01T00:00:00Z",
    )
  )
  resp = client.fetch(request)
  obj['obj'] = resp
  yield obj_store.put(obj)
```

Fig. 6. Python `fetch` step for “Rogue Zone (Airflow)” application.

(1) “variable” definitions: these map a name to a Brick query defining the context for a point and the desired engineering units for that point (if known), and aggregation function (min, max, mean, count, or raw).
(2) temporal parameters: defines the bounds on the data, desired resolution, and if we want aligned timestamps.

The output of the `fetch` component is an object providing access to the results of the Brick queries, the resulting timeseries dataframes, and convenience methods for relating specific dataframes based on the Brick context (e,g., the setpoint timeseries related to a given sensor timeseries).

Figure 6 has an example of the `fetch` component for the “Rogue Zone Detection” application. The query requests 1 year of data on air flow sensors and setpoints aggregated by mean in 10-minute buckets.

```python
def run(objid):
    data = obj_store.get(objid)
    data['cleaned_sensors'] = data['sensors'].dropna() # drop null values
    data['cleaned_setpoints'] = data['setpoints'].dropna() # drop null values
    return obj_store.put(data)
```

Fig. 7. Simple Python clean step to drop null values (periods of missing data).

```python
def run(objid):
    data = obj_store.get(objid)
    resp = deserialize(data['obj'])
    # get all the equipment we will run the analysis for. Equipment relates sensors and setpoints
    equipment = [r[0] for r in resp.query("select distinct equip from airflow_points")]
    # loop through data columns

    for idx, equip in enumerate(equipment):
        # for each equipment, pull the UUID for the sensor and setpoint
        q = """
        SELECT sensor_uuid, sp_uuid, airflow_sps.equip, airflow_sps.site
        FROM airflow_sensors
        LEFT JOIN airflow_sps
        ON airflow_sps.equip = airflow_sensors.equip
        WHERE airflow_sensors.equip = "{0}";
        """.format(equip)
        res = resp.query(q)
        if len(res) == 0:
            continue

        sensor_col = res[0][0]
        setpoint_col = res[0][1]

        # create the dataframe for this pair of sensor and setpoint
        df = pd.DataFrame([sensor_df[sensor_col], setpoint_df[setpoint_col]]).T
        df.columns = ['airflow','setpoint']
        bad = (df.airflow + 10) < df.setpoint # by 10 cfm
        if len(df[bad]) == 0: continue
        # use this to group up ranges where the sensor reads below the setpoint
        df['below_setpoint_group'] = bad.astype(int).diff().ne(0).cumsum()

        groups = df[bad].groupby('below_setpoint_group')
        for group in groups:
            print 'VAV {0} had high airflow from {1} to {2}'
              .format(eqiupment, grp[0], grp[-1]]
```

Fig. 8. Python analyze step to find periods when the measured airflow is lower than the setpoint in the Rogue Zone Airflow application.

5.3 Data Cleaning

The **clean component** is executed on the output of the fetch component. The purpose of this component is to normalize the timeseries data for the analyze component, which is executed next. Common operations in the clean component are hole filling, specialized aggregation, and data filtering. It is kept modular to facilitate the re-use of standard cleaning steps. Application developers can build their own cleaning components or leverage existing methods. A simple example of a clean component that drops missing data is presented in Figure 7.

5.4 Application Execution

The **analyze component** contains the actual application logic and is also executed for each site in the execution set. It can optionally output a result that is delivered to the final component. Figure 8 contains the logic of the "Rogue Zone Detection" application. It finds contiguous segments of time during which the measured airflow is lower than the airflow setpoint.

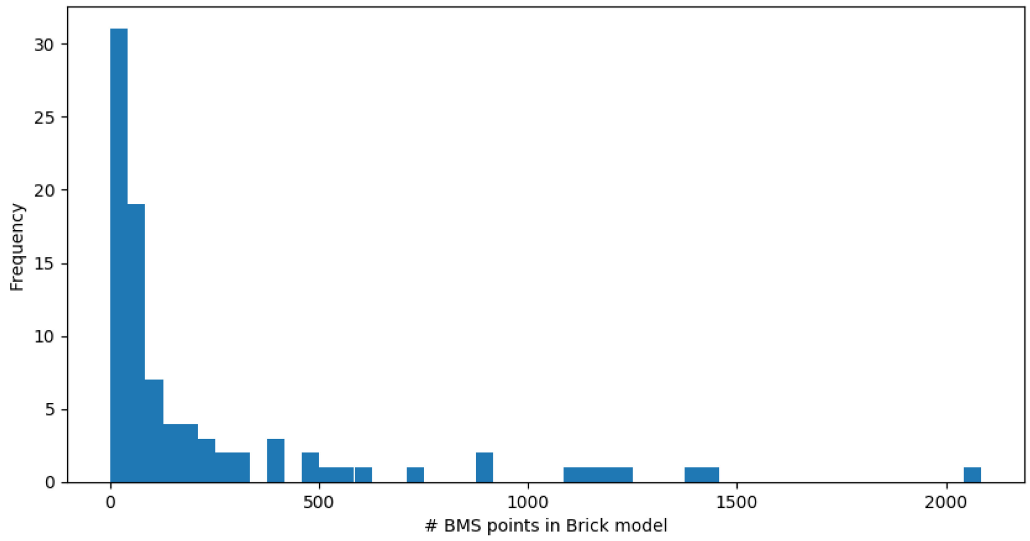


Fig. 9. Histogram of number of data streams for all sites ($\mu = 241$).

Table 2. Count of Streams and Equipment Available in the Testbed Dataset, Aggregated by Type

Temperature Sensor	7,380	Luminance Sensor	257
Occupancy Sensor	445	Pressure Sensor	148
Outside Air Temp. Sensor	362	Cloud Cover	32
Setpoints (generic)	2,331	Power Meters	77
VAVs	4,724	AHUs	467
HVAC Zones	4,887	Dampers	1,662
Non BMS Thermostats	123		

AHU and VAV totals include related equipment such as fans and pumps.

The **`aggregate` component** is an optional component that is executed once for an application, and takes as an argument the output of the `analyze` component from all sites. This is where an application can perform any final aggregation or analysis across sites.

6 THE MORTAR DATASET

As of this writing, Mortar contains 9.1 billion data points of timeseries data for 90 buildings, constituting more than 750 million hours of data. The majority of data streams in Mortar are at a 15-minute intervals, though some are more fine-grained (up to 1 second).

Figure 9 describes the distribution of the number of streams per building. Each building is accompanied by a Brick model that describes the building, its equipment and subsystems, available points, and references to timeseries data streams. The Brick models in the Mortar dataset range in size from 2,117 to 8,763 nodes. Table 2 enumerates some of the types of available points and equipment in the testbed.

Depending on what data is available, Brick models are generated from Building Information Modeling (BIM) models of buildings, architectural diagrams, site visits, and interviews. The framework for generating Brick models is open source,[7] and aims to integrate with a more mature Brick model generation framework such as Scrabble [20].

The majority of the dataset is made up of large commercial buildings belonging to a university campus. The buildings are typically used as offices, classrooms, research facilities, and health care clinics. The average building has a floor area of 70,000 sqft, 3 floors and more than 100 rooms, while the largest building is a large library with a floor area above 400,000 sq. ft. Most buildings are conditioned using large built-up HVAC systems with air handlers and local distribution boxes, controlled by building automation systems. Chilled water and hot water are produced by a central plant and distributed through large pipes to most buildings. Some additional chillers are installed in some buildings to complement the central system.

Other buildings in the dataset come from a set of independent data collecting efforts. Most of the non-campus buildings are part of an ongoing project to develop a building operating system; these are mostly small commercial buildings ranging from movie theaters to fire stations to animal shelters. Data collected includes thermostats, building meters, occupancy, temperature, illumination and humidity sensors, electric vehicle charging stations, and solar panels.

An important concern is the anonymization of the data in the testbed. Mortar anonymizes the names and locations of buildings (using nearby NOAA weather stations to indicate general location) and the names of entities in the Brick metadata model. Future releases of Mortar will support the selective deanonymization of this metadata to authorized users. Mortar does not currently anonymize the timeseries data values; because Mortar supports continuous data collection, it is difficult to apply existing best-practices such as differential privacy.

7 EVALUATION

The key contribution of this article is a testbed for the robust implementation and evaluation of portable building analytics applications. In order to evaluate the efficacy of Mortar, we have implemented a family of 11 applications. We chose these applications to exercise the ability to pull data from buildings using queries against a Brick model, and also to make use of the proposed architecture for portable applications. To test the former, we implement applications that require the use of Brick relationships to properly describe the data needed to run the application. To test the latter, we implement all applications using the decomposed architecture proposed in Section 5; some of the applications involve similar or identical logic and can re-use each other's components, using Brick queries to adjust the source data.

7.1 Applications

In this section, we provide a brief description of the applications that were used to evaluate Mortar. All applications are implemented in the Python programming language, which allows application developers to make use of the mature set of data science and statistics libraries available. A summary of the implementations of these applications is provided in Table 3. The portable implementations of these applications is released in tandem with the dataset, forming the beginning of an open library of analytics applications.

1. Baseline Calculation: This application ports an existing open-source package (LBNL-baseline[8]), which implements a baseline calculation algorithm [25], to Mortar. The package requires access to electric load data as well as outside temperature data when available, which is retrieved using a simple Brick query. Porting this application to Mortar allows us to easily compare

[7]https://github.com/gtfierro/BrickMason.
[8]https://github.com/LBNL-ETA/loadshape.

Table 3. Applications: Brick LOC and App LOC Indicate the Lines of Code Needed to Define the Brick Queries and Application Logic, Respectively

Category	Application	Brick LOC	App LOC	# sites	% coverage
Measurement, Verification & Baselining	Baseline Calculation	3	120	33	37%
	EUI Calculation	10	100	7	8%
	HVAC Energy Disaggregation	14	124	14	16%
	Thermal Model Identification	17	339	17	19%
Fault Detection & Diagnosis	Rogue Zone Temperature	15	104	56	62%
	Rogue Zone Airflow	7	98	3	3%
	Baseline Deviation	3	204	14	16%
	Stuck Damper Detection	8	91	30	33%
	Obscured/Broken Lighting Detection	11	100	2	2%
Advanced Sensing	Virtual Coil Meter	14	150	60	67%
	Chilled Water Loop Total Electrical Consumption	17	160	15	17%

"% coverage" is what proportion of the testbed's buildings qualified for that application; the corresponding number of buildings is in the "# sites" column.

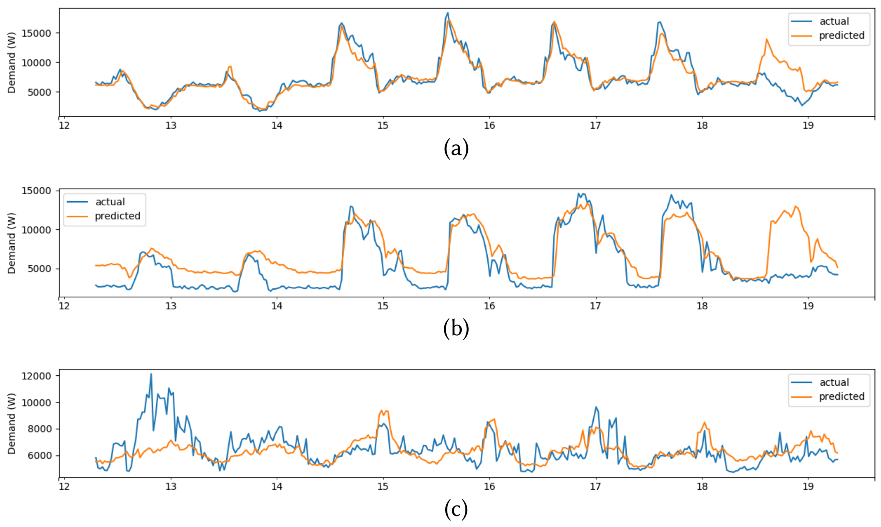


Fig. 10. **Baseline prediction app:** Predicted baseline using [25] plotted against actual building energy consumption.

a predicted baseline with historical data across a variety of sites with differing climates, construction, and usage patterns. Some results of this application are shown in Figure 10.

2. Baseline Deviation: We made use of the modular implementation of the Baseline Calculation application to implement this application. In particular, we modified the `analyze` component from the Baseline Calculation application (keeping the rest of the stages) to compare measured energy consumption with the predicted baseline to identify periods of abnormally high consumption.

3. Energy Usage Intensity (EUI) Calculation: The calculation of EUI involves dividing the yearly energy consumption of a building in kilowatt-hours (kWh) by the total floor area of that building. This metric is often used to benchmark buildings. Our implementation of EUI uses a Brick query to discover what kind of building energy consumption data is available. In the current dataset, sites report consumption either through an instantaneous demand meter such as a Rainforest Eagle[9] or through an energy bill such as PG&E's "Share My Data" service. Availability of meter data can be determined through a Brick query, and the list of sites with building meters can be retrieved using the Mortar `Qualify` API method. The floor area of a building can also be captured in the Brick model using the extensions from Table 1. The logic of the application retrieves a year of data, performs the appropriate calculation (either a conversion from instantaneous power data to kWh or a simple sum of regular kWh readings), and divides it by the total floor area as retrieved from the Brick model.

4. HVAC Energy Disaggregation: This application estimates the energy consumption of equipment by correlating changes in electrical demand (through the use of a building-level meter or submeter) with changes in HVAC equipment state. In the case of binary state points (e.g., `Compressor On Off Status`), the disaggregation application can estimate the total energy usage of a piece of equipment. The disaggregation application could then make use of any available weather and internal temperature data in order to identify any change in efficiency in the equipment's operation corresponding to changes of environmental variables. Our implementation leverages the Brick class hierarchy to find all points of type `Status` that relate to equipment in the building; the resulting queries can easily be refined to look only at specific classes or instances of equipment such as only single-stage or multi-stage equipment.

5. Thermal Model Identification: This application trains a zone-level thermal model using zone- and room-level (using additional sensors, if present) temperature sensing, outside air temperature, cloud coverage, HVAC equipment state data and, when available, adjacency information. The implementation uses Brick queries to determine what information is available such as the number of zones, the temperature sensing in those zones, HVAC equipment conditioning those zones, and available weather data.

6. Rogue/Critical Zone Temperature: This application detects rogue zones, which are thermal zones whose temperature consistently measures outside of the setpoint temperature band; this can be caused by thermal loads larger than design conditions, incorrect setpoints, and/or broken sensors or equipment [9]. For critical zones, it indicates that the air temperature leaving the AHU should be lower/higher. In large HVAC systems, these zones can cause an increase in the energy use of the entire air handling units, due to increased reheat in the other zones. The detection of rogue zones can also be used to improve control sequences.

7. Rogue/Critical Zone Airflow: This application detects rogue zones whose airflow is consistently below the airflow setpoint. It uses Brick models to pull in additional context about the building. The airflow setpoints and sensors can be related through the HVAC infrastructure to the zones containing the rooms affected by the reduced airflow. This information is typically used as an input to the duct static pressure controller on the air handling unit serving these zones.

8. Stuck Damper Detection: This application performs the common fault-detection task of identifying dampers whose position has not changed in weeks or months. Our implementation automates the task of finding the damper position status streams and identifies the HVAC zones and rooms affected by the damper. For zones with discharge airflow sensors the application can also test if the damper appears to work correctly, but the amount of air supplied does not change with its position. This can indicate a faulty sensor or a broken linkage between actuator and the physical damper.

9. Obscured Lighting Detection: This application groups lighting status points for a lighting zone with luminance sensors in that zone in order to build a correlation between time, lighting

[9]https://rainforestautomation.com/rfa-z114-eagle-200/.

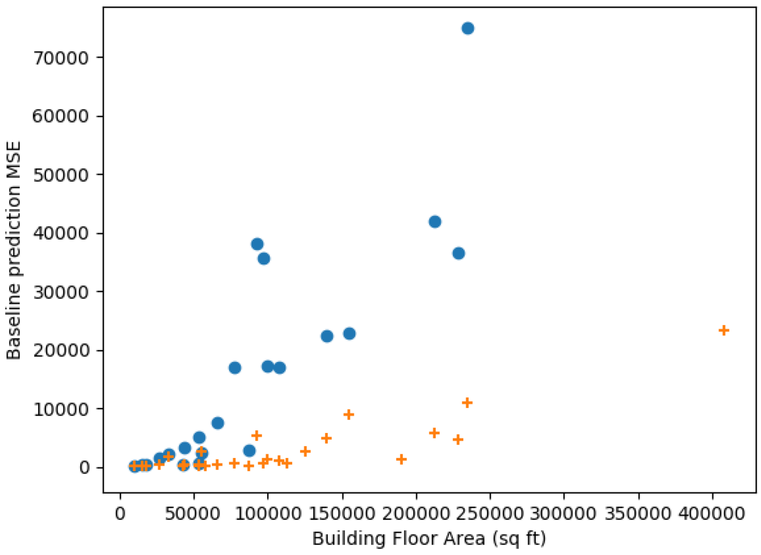


Fig. 11. Comparing the mean squared error of building energy consumption prediction using the baseline from [25] trained and evaluated on summer months (+) versus winter months (●).

status, and luminance. Deviations from this model can be used to identify broken or obscured fixtures and luminaires.

10. Virtual Coil Meter: This application estimates the amount of heat energy used by a heating or cooling coil by performing a calculation over upstream and downstream air temperature sensors, air flow sensors, and the position of the valve in the coil [33]. The portable implementation of this application discovers these points through querying the Brick model for a building for their relationships to each other and the building's HVAC system.

11. Chilled Water Loop Total Electrical Consumption: This application examines all equipment on a chilled water loop and sums the electrical consumption of any component found on that loop. Assembling this collection of equipment and related building points without a Brick model involves a high degree of manual effort that does not carry over when porting the application to other buildings.

7.2 Results and Discussion

The open building testbed presents an opportunity to compare multiple buildings in a portfolio, measure the portability of an application, measure the accuracy or performance of an application, compare the performance of similar applications, and automate tedious re-configuration when porting an application to different buildings. To demonstrate these features, we examine the results of running a few applications from Table 3 against the Mortar testbed.

First, we examine the accuracy of the building energy estimation baseline algorithm from [25] across winter and summer seasons (Figure 11). In winter, the mean squared error of the baseline prediction compared to a week's ground truth increases with the floor area of the building. The positive correlation between square footage of the building and the error in energy prediction suggests that larger buildings have more variability in their loads than the baseline estimation algorithm is able to account for. The higher error in the winter months compared to summer months additionally suggests a seasonality to this error. A more thorough evaluation of the baseline estimation algorithm is beyond the scope of this article, but this initial result demonstrates the value of being able to evaluate an analytics algorithm over a large corpus of buildings.

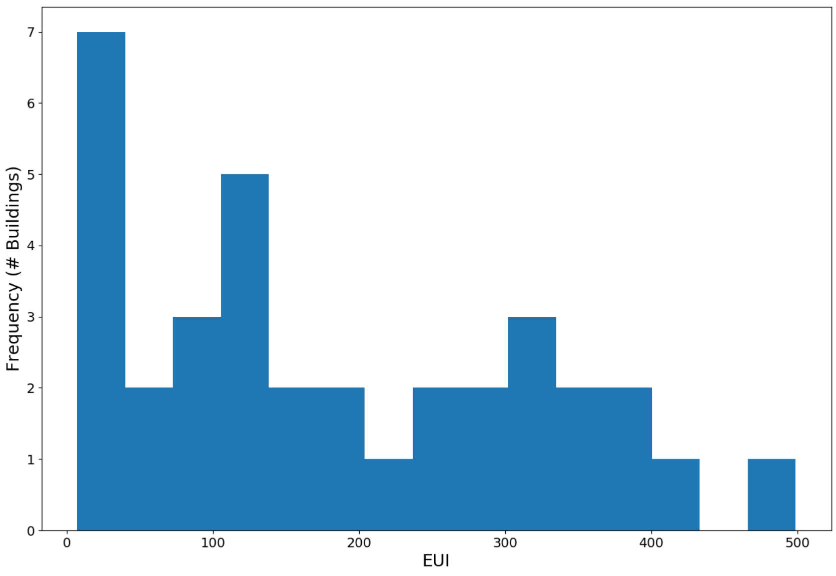


Fig. 12. Distribution of EUI across the Mortar portfolio.

The Thermal Model application is a good example of how applications can change their behavior based on the information available in a building. The `qualify` step looks for buildings with thermostats and RTUs with or without room-level temperature sensors. In the case when only thermostats are available, the application can provide a zone-level thermal model that runs on 17 buildings from the testbed. For buildings where room-level temperature sensors are available, the application takes those into account to provide a more fine-grained room-level thermal model, but this only runs on three buildings from the testbed. If the application were to be extended with a thermal modeling approach for AHU-based systems with heating and cooling coils, then the thermal model application would operate over all 90 buildings.

The Mortar testbed also presents the opportunity to examine building performance across a portfolio. In Figure 12, we plot the distribution of EUI over the whole testbed portfolio; users could adjust the Energy Use Intensity application `qualify` step to filter by different building properties, such as climate and square footage. The inclusion of building properties in the Brick model for each building facilitates this flexible portfolio management.

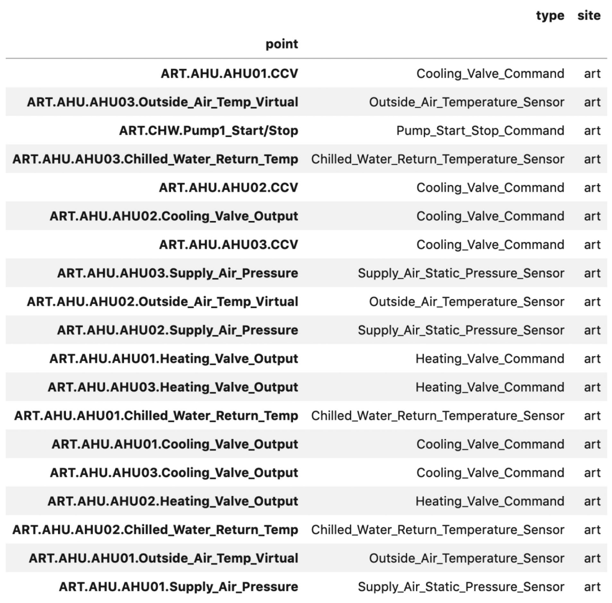
Mortar also lets users skip tedious configuration steps. Without the use of a Brick model, assembling the points for the Chilled Water Loop Total Electrical Consumption application requires site-specific knowledge as to what power meters are present in the building, what equipment they measure, and whether or not that piece of equipment is related to a particular chilled water loop. Performing this assemblage over a large building portfolio can be unmanageable.

8 APPLICATION DEVELOPMENT CASE STUDIES

To qualify the experience of using the Mortar platform, we asked several researchers to implement and describe several building-related analytics applications.

8.1 Application: Available Brick Points

Data discovery is an important facet of the Mortar platform. While users who are familiar with Brick can execute queries to determine which data sources are available for a particular building, it can be helpful to newer users to see a list of all available data sources.



point	type	site
ART.AHU.AHU01.CCV	Cooling_Valve_Command	art
ART.AHU.AHU03.Outside_Air_Temp_Virtual	Outside_Air_Temperature_Sensor	art
ART.CHW.Pump1_Start/Stop	Pump_Start_Stop_Command	art
ART.AHU.AHU03.Chilled_Water_Return_Temp	Chilled_Water_Return_Temperature_Sensor	art
ART.AHU.AHU02.CCV	Cooling_Valve_Command	art
ART.AHU.AHU02.Cooling_Valve_Output	Cooling_Valve_Command	art
ART.AHU.AHU03.CCV	Cooling_Valve_Command	art
ART.AHU.AHU03.Supply_Air_Pressure	Supply_Air_Static_Pressure_Sensor	art
ART.AHU.AHU02.Outside_Air_Temp_Virtual	Outside_Air_Temperature_Sensor	art
ART.AHU.AHU02.Supply_Air_Pressure	Supply_Air_Static_Pressure_Sensor	art
ART.AHU.AHU01.Heating_Valve_Output	Heating_Valve_Command	art
ART.AHU.AHU03.Heating_Valve_Output	Heating_Valve_Command	art
ART.AHU.AHU01.Chilled_Water_Return_Temp	Chilled_Water_Return_Temperature_Sensor	art
ART.AHU.AHU01.Cooling_Valve_Output	Cooling_Valve_Command	art
ART.AHU.AHU03.Cooling_Valve_Output	Cooling_Valve_Command	art
ART.AHU.AHU02.Heating_Valve_Output	Heating_Valve_Command	art
ART.AHU.AHU02.Chilled_Water_Return_Temp	Chilled_Water_Return_Temperature_Sensor	art
ART.AHU.AHU01.Outside_Air_Temp_Virtual	Outside_Air_Temperature_Sensor	art
ART.AHU.AHU01.Supply_Air_Pressure	Supply_Air_Static_Pressure_Sensor	art

Fig. 13. List of `Points` and their types in the Brick schema for a particular site in the Mortar dataset.

This application queries a site's Brick model for the names of all instances of the `Point` class, which correspond to the available data sources. The returned list of data sources—organized by type—allows users to see what types of timeseries data exist for a particular site.

The application can fetch all the points and corresponding types for a specific site or for all sites in the Mortar dataset. This output is saved as a CSV file. Figure 13 contains typical output of this app.

8.2 Application: Simultaneous Heating/Cooling AHUs

Simultaneous heating and cooling is one of the most common kinds of faults in AHUs [12, 22, 31, 36]). AHUs have a heating coil valve and a cooling coil valve that condition (heat or cool) the air distributed throughout a building; some models additionally contain valves that control the volume of outside air to be mixed. The AHU adjusts the degree of heating or cooling by opening and closing the valves; this is typically measured in percent open (0–100%). Having both valves open at the same time is a waste of energy (except for dehumidification purposes [22]).

This Mortar application performs detection of simultaneous heating and cooling in AHUs using the rule-based fault diagnosis algorithms from [12, 36]. The application uses a single Brick query to fetch all`Brick:Cooling_Valve_Command` and `Brick:Heating_Valve_Command` points for all AHUs across the Mortar testbed; these correspond to the control commands (0–100%) sent to the cooling coil valve and the heating coil valve, respectively.

The application uses the Mortar `fetch` command to retrieve this data, and executes the analysis to identify AHUs that have periods of time during which both the heating and cooling valve commands are greater than 0. The application outputs the name of the AHU and all time ranges containing suspected simultaneous heating and cooling. Figure 14 shows the heating and cooling valve commands for one AHU in a site in the Mortar dataset during a week in summer of 2017. It can be seen that the heating coil value is almost always completely open, and the cooling valve is never closed; therefore, this is an example of simultaneous heating and cooling.

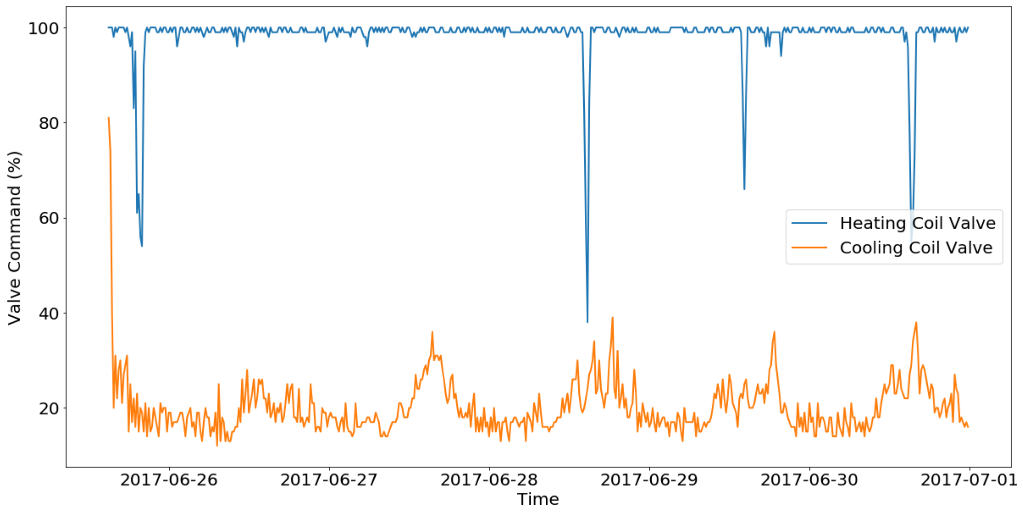


Fig. 14. Heating coil and cooling coil valve commands over time for an AHU in a Mortar site demonstrating simultaneous heating and cooling.

At the time of writing, executing this application returned data for 54 qualified sites. Execution took less than 50 seconds to analyze data for the 3-month period "2017-01-01 00:00:00" to "2017-04-01 00:00:00" across all qualified sites.

8.3 Application: Energy Consumption Baselines

Building energy consumption baselines are an important component of Measurement and Verification, and can help facility managers understand the impact of energy efficiency measures and programs. There are many existing techniques for generating baselines and predicting energy consumption; however, it has been difficult to compare these techniques across diverse sites due to lack of access to data and the effort involved in implementing and porting the techniques to each site.

The Mortar platform enables large-scale comparisons of different approaches. We explore one such approach here, but other baseline algorithms have been implemented over Mortar.[10] The approach implemented in this application uses regression analysis, which has shown promising results in predicting energy consumption [16] while being relatively easier to implement.

To compute the energy consumption baseline for a given target date, the application first pulls all relevant data prior to that date corresponding to the chosen feature set which includes outdoor air temperature, time of day, day of week, and past energy consumption. The application fits the regression and outputs the model accuracy (adjusted R-squared) for a target date and produces plots of predicted consumption against true consumption for each site (example: Figure 15). The target date may be chosen to compare against other baselining methods, or it may be chosen to estimate the impact of an energy efficient control scheme deployed on that day (e.g., to estimate the amount of energy saved).

Mortar provides a large amount of building data which makes it easy to calculate energy baselines on a diverse set of buildings, which allows a user to gain substantial insights into which types of baselining techniques work on which kinds of buildings.

[10]https://github.com/SoftwareDefinedBuildings/mortar-analytics/tree/master/baselines/energy_consumption_baseline.

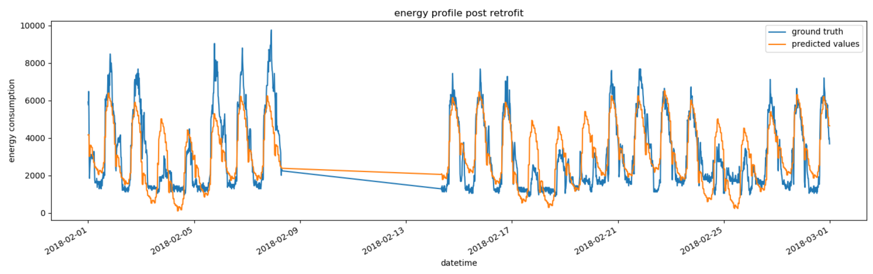


Fig. 15. Energy profile of a site post retrofit.

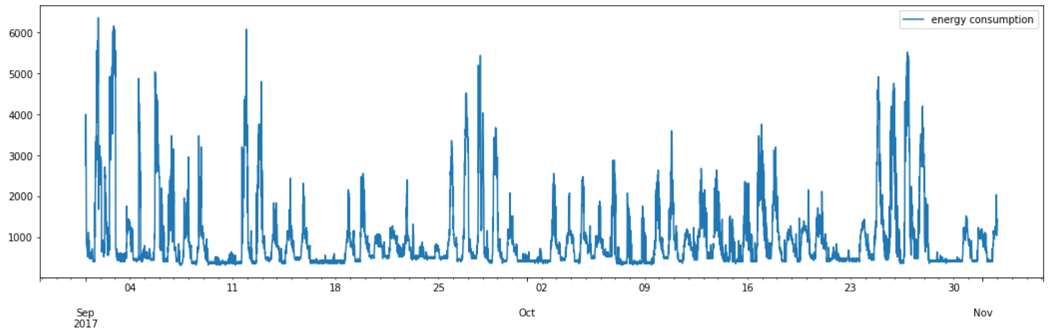


Fig. 16. Energy consumption of building.

8.4 Application: Building Occupancy and Energy Correlation

Building managers generally schedule a building's HVAC system to run on a fixed time schedule (e.g., 9AM–5PM) [17] with the assumption that this period corresponds to operating hours, or at least when the building is expected to be occupied. However, the actual occupied periods may not align with this schedule (e.g., university buildings are often active from 11AM to 9PM) or there may be an exception to the schedule (such as weekends or federal holidays). These are periods when the building is unoccupied but the HVAC system is still running. The mismatch between the programmed occupancy schedule and the actual occupancy of a building is a potential source of both occupant discomfort and energy wastage [30, 41].

This application calculates the energy consumed by each building when it is occupied. The application uses two Mortar queries to fetch the occupancy data and energy consumption for buildings that have both features available. The `Qualify` API method lets Mortar filter out the buildings without the required data.

The application produces plots showing the correlation between the energy consumption and the recorded occupancy for each building. Figures 16 and 17 show the respective plots of a single site. This data can be used to compute the amount of energy expended when the building is occupied versus when it is unoccupied.

8.5 Application: Inefficient Zone Detection

This application detects HVAC zones with inefficient control strategies by finding HVAC zones that were both heated and cooled in the same time interval. Heating and cooling in the same interval does not necessarily point toward malfunctioning equipment in the zone, but it does suggest that there are possible changes that can be implemented to improve the operating efficiency.

The application uses a single `Fetch` method call to retrieve thermostat data—heating/cooling state, indoor air temperature, and heating/cooling setpoints (abbreviated HSP and CSP)—along

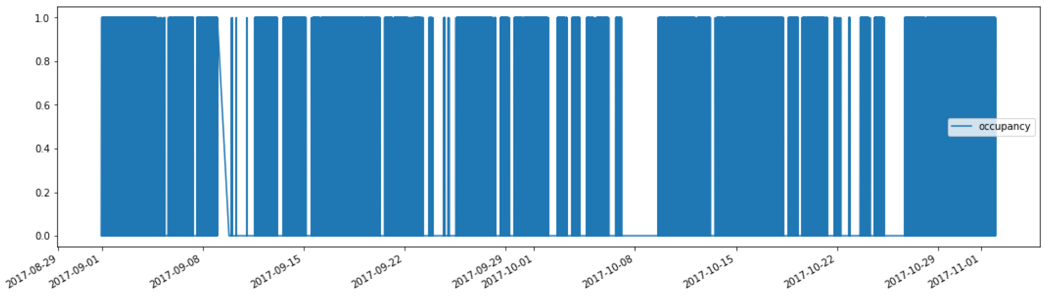


Fig. 17. Occupancy of building.

Table 4. Thermostat State Enumeration in the Mortar Platform

Value	Thermostat State
0	Off
1	Stage 1 Heating
2	Stage 1 Cooling
3	Fan only
4	Stage 2 Heating
5	Stage 3 Cooling

with the thermostat's location and the name of the HVAC zone it is controlling. Only the heating/cooling state of the thermostat is required for the analysis, but having the setpoints and indoor air temperature data available helps provide more context to the detected event: for example, to determine if the inefficiency was due to a change in setpoints or because there was an observed rapid change in temperature.

In the Mortar platform, each thermostat can be in one of the states listed in Table 4. Because of this quantization, Mortar can return uninterpretable data for the heating/cooling state if the `mean` or `median` aggregation functions are used. The application uses the `max` aggregation function, which preserves the enumerated integer values.

The application operates by scanning through the retrieved data using a configurable time interval. It uses the above enumeration of states to decide if in any interval, the thermostat has been in both heating (1 or 4) and cooling (2 or 5) states. The application outputs the range of each interval that meets this condition, with additional context: the HVAC zone, room containing the thermostat, the thermostat setpoints, and the percentage of the time the thermostat was in either the heating or the cooling state.

This approach can be augmented with logic to filter out different faults: for example, zones that do not meet a minimum threshold of heating or cooling duration, or identifying ranges of time when two different zones in a building are heating and cooling. Table 5 and Figure 18 show an example of a zone that has been in both heating and cooling states in the same hour for 3 consecutive hours on a day (2018-12-17 20:00:00 to 22:00:00) in winter. Figure 18 shows that the inefficiency is due to setpoints being changed to induce cooling.

8.6 Application: Baseline Prediction and Demand Response Event Evaluation

Demand Response (DR) event programs encourage customers to shift energy consumption away from peak hours, usually through increased energy prices. For building managers and utilities, it

Table 5. A Subset of the Output from the Inefficient Zone Detection App

time	*% heating*	*% cooling*	*min HSP*	*min CSP*	*max HSP*	*max CSP*
20:00:00	30.00	40.00	60	63	66	69
21:00:00	16.67	16.67	61	64	66	69
22:00:00	1.67	25.00	60	63	64	69

time: Hour of Day on 2018-12-17 when the Room 102 (Zone HVAC_Zone_Rm2) was Both Heated and Cooled; *% Heating*, *% Cooling*: Percentage of the Time in the Hour it was in Heating and Cooling States; *min HSP*, *min CSP*, *max HSP*, *max CSP*: Minimum and Maximum Values of Heating and Cooling Setpoints in Fahrenheit During that Hour.

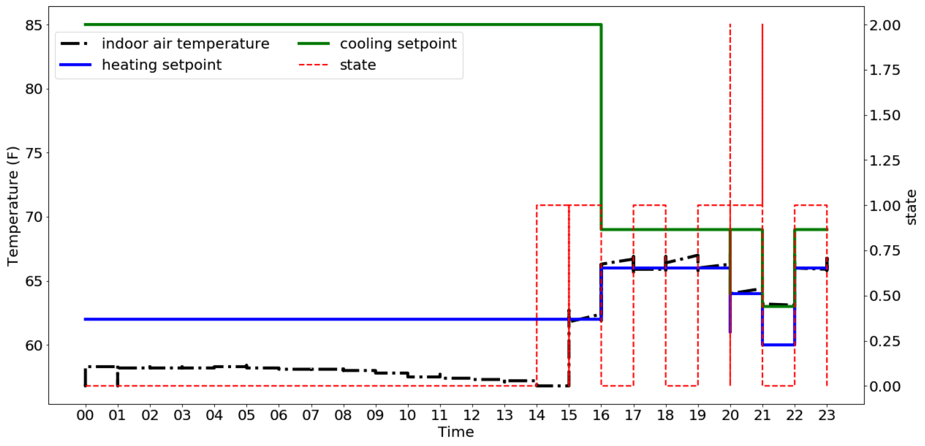


Fig. 18. Thermostat data for 2018-12-17 showing the change in the state, the setpoints, and the indoor air temperature.

is useful to quantify energy and cost savings during a DR event. Calculating these metrics requires a baseline, which can predict energy consumption during the day of the DR event assuming no DR program is applied. The calculated baseline can be compared with actual energy consumption to estimate the energy and cost savings during the DR event.

This application takes as input a site name, DR event dates, and a window of time to train the baseline prediction model on. It outputs the model test error, baseline predictions (in 15-minute intervals), and estimated energy and dollar savings for each DR event.

The application performs data collection, feature engineering, model selection, and DR event evaluation (summarized in Figure 19). The data collection stage performs two queries to fetch 15-minute interval data over a configurable window: one to fetch energy from points of type `brick:Building_Electric_Meter` and one to fetch outdoor air temperature from points of type `brick:Weather_Temperature_Sensor`.

The application extends a baseline calculation algorithm [25] with temperature change over time. The resulting features are as follows:

— Time-of-week: An indicator variable $a_i \in \{0, 1\}$ for the ith 15-minute time interval of the workweek, $i \in [0, 479]$.
— Piece-wise temperature: Temperature split into intervals. Given a temperature in degrees Fahrenheit t, let $f_j = \max(\min(t - C_j, C_{j+1} - C_j), 0)$, where $C = [0, 40, 50, 60, 70, 80, \infty]$, $j \in [0, 5]$ (six features total).

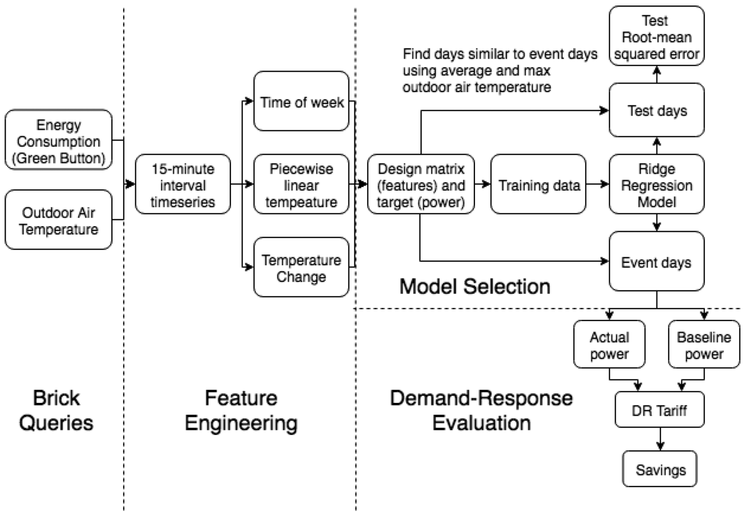

Energy Consumption (Green Button)
Outdoor Air Temperature
15-minute interval timeseries
Time of week
Piecewise linear tempeature
Temperature Change
Design matrix (features) and target (power)
Training data
Ridge Regression Model
Test days
Test Root-mean squared error
Event days
Find days similar to event days using average and max outdoor air temperature
Model Selection
Actual power
Baseline power
DR Tariff
Savings
Brick Queries
Feature Engineering
Demand-Response Evaluation

Fig. 19. Implementation of the DR event baseline prediction and evaluation.

— Temperature Change: Change in temperature from the previous timestep, measured as $t_k - t_{k-1}$.

After feature engineering, the application chooses test days and removes them from the training set. In order to evaluate model accuracy on DR event days (assuming no program applied), the application finds days that are "similar" to event days (typically other hot days). First, the application identifies the mean and max temperature of DR event days and finds the median of these two values. It then searches its data for all dates with mean and max temperatures above these cutoffs. Finally, it randomly selects half of these "qualified" dates for testing, and keeps the remaining dates in the training set to avoid extrapolation. DR event days are also removed from the training set. The app then fits the training set with an l_2-regularized linear regression model, using cross-validation to pick the regularization term.

The application uses the trained model to predict the baseline energy consumption on the test dates and the DR event dates. The application first finds the "best" test dates using the RMSE between the prediction and actual consumption, then compares the predicted energy consumption for the "best" test day with actual consumption on DR event dates and uses the difference between them to compute energy and dollar savings.

The application was tested on 10 buildings in California pulled from the Mortar dataset. Standard errors ranged from 1.34 kW to 5.46 kW, with an average of 3.48 kW and a median of 3.27 kW. The distribution of errors and example feature weights are displayed in Figure 20. Examples of baseline predictions are displayed in Figure 21.

Mortar makes it possible to develop this application for a variety of buildings. It is simple to define the time-series to query, whether it comes from a specific kind of data source (Building Electric Meter) or a class of data sources (Temperature Sensors). It also accommodates a variety of time-series intervals and aggregations, which makes it intuitive to find test days that were similar to DR event days by using the MAX and MEAN aggregations over 24-hour time intervals. Access to the large array of buildings allows the application to evaluate the regression model in a variety of contexts, and potentially find a more predictive subset of features for each building.

Our experience writing this application suggests that Mortar can improve its handling of missing data. Higher standard errors came from buildings missing temperature data. Additionally, more

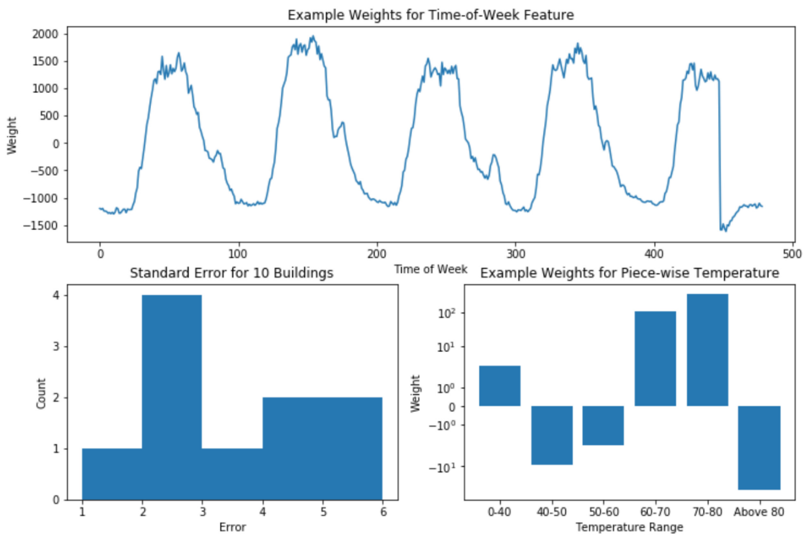


Fig. 20. The bottom-left histogram displays the test standard errors for each of the 10 buildings. The top and bottom-right plots display example weights for a baseline model trained for an office building. The model captures the daily period of demand, as well as the non-linear relationship between temperature and demand described in [25]. The weight for temperature change was −39.8.

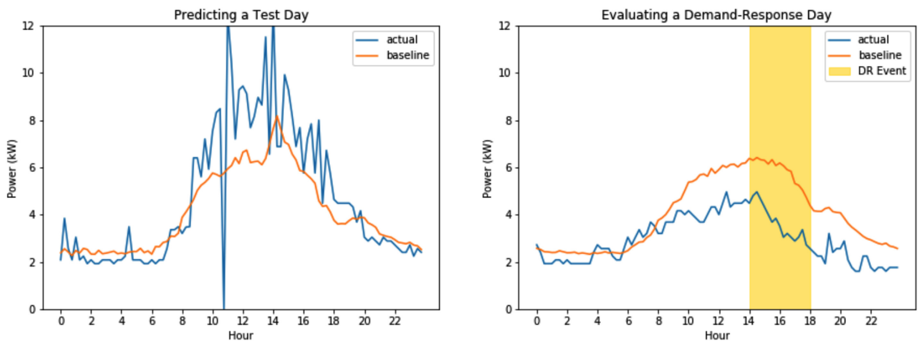


Fig. 21. The left plot displays the prediction of energy consumption in an office building on a test day. The right plot displays a DR event which produced desired results when compared to the baseline—an estimated $10 of savings and 7 kWH reduced energy consumption during the event.

metadata on the weather temperature sensors would be helpful; frequently, the query returned multiple weather temperature sensors and it was unclear which sensor was the most accurate or closest to the building.

8.7 Application: Data-Driven Black-Box Thermal Modeling

Developing control algorithms to improve the energy efficiency of HVAC systems requires indoor temperature forecasting models. Most state-of-the-art thermal models rely on gray-box and white-box models that have a notion of the structure and the layout of a building [2]. Because gray- and white-box models are site-specific, it is difficult to scale their development to support widespread predictive HVAC control. This application investigates the efficacy of a data-driven black-box thermal model that can generalize across multiple buildings. While similar models have

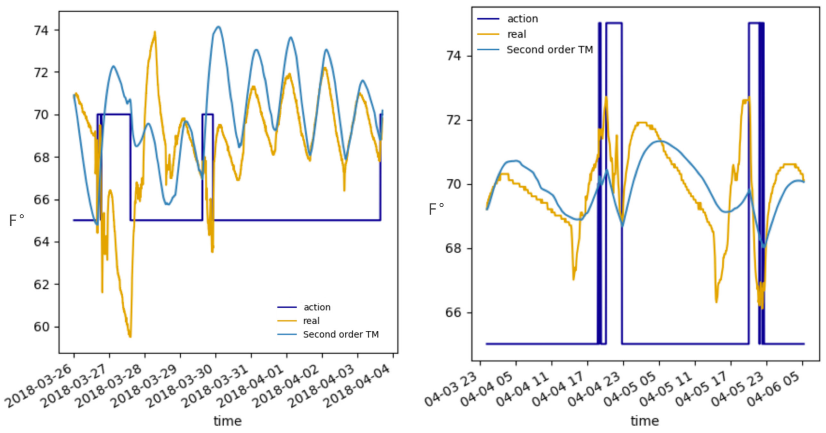


Fig. 22. Forecasting results for a sample site. The "real" line is the true indoor temperature. "Second-order TM" lines are the forecasting results. The "action" line indicates for the left (right) image whether or not heating (cooling) happened in the given zone.

been proposed before [34], to our knowledge they were not tested against an extensive testbed such as provided by Mortar. Future work is to reproduce existing work on the Mortar testbed.

The application utilizes a second-order regression model:

$$\begin{aligned} T_{next} = &T_{in} + a_{heating}c_{heating}(T_{in} - b_{heating}) + a_{cooling}c_{cooling}(T_{in} - b_{cooling}) \\ &+ c_{out}(T_{out} - T_{in} - b_{out}) + c_{bias}(T_{in} - b_{bias}) \\ &+ \sum_{i}^{num\_zones} c_{zone_i}(T_{in} - T_{zone_i} - b_{zone_i}) \\ &+ c_{prev}T_{prev} \end{aligned}$$

with

$$\begin{aligned} T_{next} &= \text{the predicted indoor temperature } \tau \text{ minutes into the future,} \\ T_{in} &= \text{the current indoor temperature in the given HVAC zone,} \\ T_{out} &= \text{the current outdoor temperature,} \\ T_{zone_i} &= \text{the indoor temperature of the other zones in the building,} \\ T_{prev} &= \text{the temperature } \tau \text{ minutes ago.} \end{aligned}$$

The application uses Mortar to fetch timeseries data for each of the parameters described in the thermal model equation. The data collected spans 200 days of training and 50 days of testing data for each feature, totaling over 60,000 data points at the time of writing. Figure 22 contains forecasting results for the trained thermal model.

The application uses regression models rather than non-linear models because (1) regression models retain some interpretability over the learned coefficients, (2) linear models are needed for applying Linear Programming to building control, and (3) more complex models did not demonstrate better performance across buildings.

One major benefit of Mortar is that developers can quickly and easily explore data from various buildings. This ability to cross-reference data across buildings motivated and justified the use of a second-order model in this application. Various higher order effects such as temperature decreas-

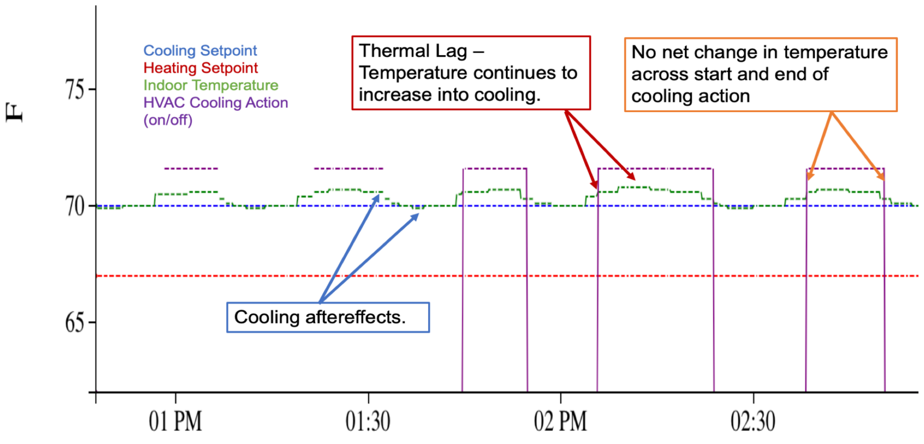


Fig. 23. Three of the core issues encountered in indoor temperature data.

```
SELECT ?tstat ?temp WHERE {
    ?tstat rdf:type brick:Thermostat .
    ?tstat bf:controls/bf:feeds mybuilding:HVAC_Zone_1 .
    ?tstat bf:hasPoint ?temp .
    ?temp rdf:type brick:Temperature_Sensor  .
};
```

Fig. 24. Brick query to retrieve HVAC zone temperatures.

ing even though cooling has ended are visible in the sample data in Figure 23. With Mortar, we were able to easily compare these effects across buildings to confirm that they were universal.

The Brick ontology made it easy to retrieve relevant data streams for developing the thermal model. For example, historic HVAC zone temperatures and cooling/heating actions are both in a natural relationship with a thermostat, which is related to the HVAC zone. With this understanding, we create the query in Figure 24 for HVAC zone temperatures. To then retrieve action data, we simply replace `brick:Temperature_Sensor` with `brick:Thermostat_Status` in the given query. The application embeds this query in a Mortar `Fetch` call to retrieve data for a specified time period.

9 RELATED WORK

Existing building benchmarking datasets (Table 6) are static, sparse, and/or lack sufficient context to implement potentially complex analytics. Further, there is no standard adopted naming scheme to tie these datasets together, which hinders the portability of any application developed against a single platform. In contrast, Mortar provides datasets that are continuously updated, and well annotated. Further, many of the existing datasets can be easily integrated in Mortar by representing their metadata using a Brick model and ingesting any included timeseries data. Executing and automating this process is the subject of future work.

Also related are a family of analytics platforms for building data and the smart grid. Commercial platforms like Skyspark [35] and BuildingOS [24] offer a mixture of pre-built standard analytics and composable rules and operators for user-defined analytics. Some of these platforms require analytics to be implemented in a proprietary or custom programming language, which limits the types of tools that can be used to develop analytics. There are also several academic building operating systems that implement building and smart-grid analytics [3, 10, 11, 39].

Table 6. Some Open Datasets of Building Data

Name	Year	Available Data	Updates	Metadata
Building Genome Project [26]	2017	non-residential building electric meter data	rare	building typology
CBECS [13]	2016	building energy usage, systems, and equipment survey	4–10 year cycle	building typology
NILMTK [7]	2014	high-frequency building electric meter data, equipment state	rare	labels and electrical subsystem
Smart* [6]	2012	electrical meter data, temperature and humidity sensors, applicance state for residential buildings	yearly releases	labels only
REDD [21]	2011	full building and device-level electrical meter data	none	electrical subsystem
BASE [40]	1995	temperature and humidity sensors, HVAC equipment and sensors	none	building typology, HVAC system
GREEND [28]	2013	device-level electrical meter	none	appliance labels
Pecan St Dataport [32]	2011–2018	building and appliance-level electric, gas, and water data for residential buildings	unspecified	building typology, equipment labels
REFIT [29]	2013–2015	high-frequency electric building and appliance-level meter data	none	building typology, appliance labels

"Labels only" means the dataset's metadata only identifies a timeseries with a label and little or no contextual metadata. "Building typology" means the dataset's metadata only deals with high-level information about the building such as occupancy class, climate and floor area.

The proposed portable analytics applications architecture enables users to develop applications (or port existing ones) using standard, open-source tools and frameworks, and integrate these applications into the Mortar platform. Further, Mortar allows users to add their own buildings to the testbed, and will assist them with the creation of the Brick model. Documenting and automating this process is the subject of future work.

10 CONCLUSION AND FUTURE WORK

In this article, we have presented the design and implementation of an open testbed for portable building analytics (Mortar), a diverse corpus of timeseries data and Brick models spanning multiple building classes (Mortar dataset), and an architecture for implementing portable analytics applications (Mortar portable applications). Mortar utilizes and extends Brick, a graph-based metadata model, to describe the relationships between data streams and to facilitate application portability. We have developed 11 applications from the literature to evaluate Mortar. The evaluation shows that Mortar can run a single application against multiple buildings to (a) compare the performance of multiple buildings in a portfolio (EUI score), (b) measure and improve application coverage across buildings, and (c) measure the application performance or accuracy (e.g., mean error). Further, we have showed that Mortar allows application developers to significantly reduce the manual labor involved in developing and deploying an application against multiple buildings, or compare the performance of two algorithms for a given application. Our goal is that Mortar will be used as a standard benchmarking platform for implementations of new and established analytics algorithms, leading to a new generation of robust, portable analytics with reproducible evaluations.

There are several areas of future work that will improve the usability and security of Mortar. Firstly, we are working on integrating privacy-preserving techniques for sharing continuously updated timeseries data. We are also improving the process for integrating new sources of timeseries data and metadata into the testbed. Mortar is available at mortardata.org.

ACKNOWLEDGMENTS

The opinions expressed belong solely to the authors, and not necessarily to the authors' employers or funding agencies.

REFERENCES

[1] 2018. Project Haystack. Retrieved 01 April, 2018 from http://project-haystack.org/.

[2] Abdul Afram and Farrokh Janabi-Sharifi. 2014. Review of modeling methods for HVAC systems. *Applied Thermal Engineering* 67, 1-2 (2014), 507–519.

[3] Yuvraj Agarwal, Rajesh Gupta, D. Komaki, and Thomas Weng. 2012. Buildingdepot: An extensible and distributed architecture for building data storage, access and sharing. *BuildSys' 12*, 64–71. DOI: https://doi.org/10.1145/2422531.2422545

[4] Michael P. Andersen and David E. Culler. 2016. BTrDB: Optimizing storage system design for timeseries processing. Section 3 (2016), 39–52.

[5] B. Balaji, A. Bhattacharya, G. Fierro, J. Gao, J. Gluck, D. Hong, A. Johansen, J. Koh, J. Ploennigs, Y. Agarwal, M. Berges, D. Culler, R. Gupta, M. B. Kjærgaard, M. Srivastava, and K. Whitehouse. 2016. Brick: Towards a unified metadata schema for buildings. In *Proceedings of the 3rd ACM Conference on Systems for Energy-Efficient Built Environments, (BuildSys'16)*. DOI: https://doi.org/10.1145/2993422.2993577

[6] S. Barker, A. Mishra, D. Irwin, E. Cecchet, P. Shenoy, and J. Albrecht. 2012. Smart*: An open data set and tools for enabling research in sustainable homes. *SustKDD*August (2012), 6. DOI: https://doi.org/adf

[7] Nipun Batra, Jack Kelly, Oliver Parson, Haimonti Dutta, William Knottenbelt, Alex Rogers, Amarjeet Singh, and Mani Srivastava. 2014. NILMTK: An open source toolkit for non-intrusive load monitoring. In *Proceedings of the 5th International Conference on Future Energy Systems (e-Energy'14)*, 265–276. DOI: https://doi.org/10.1145/2602044.2602051arxiv:1404.3878

[8] Vladimir Bazjanac and Drury B. Crawley. 1999. Industry foundation classes and interoperable commercial software in support of design of energy-efficient buildings. In *5th IBPSA*April (1999), 7. http://www.ibpsa.org/%5Cproceedings%5CBS1999%5CBS99_B-18.pdf.

[9] Arka A. Bhattacharya, Dezhi Hong, David Culler, Jorge Ortiz, Kamin Whitehouse, and Eugene Wu. 2015. Automated metadata construction to support portable building applications. *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments (BuildSys'15)*, 3–12. DOI: https://doi.org/10.1145/2821650.2821667

[10] Bin Cheng, Salvatore Longo, Flavio Cirillo, Martin Bauer, and Ernoe Kovacs. 2015. Building a big data platform for smart cities: Experience and lessons from Santander. In *Proceedings of the 2015 IEEE International Congress on Big Data (BigData Congress'15)*, 592–599. DOI: https://doi.org/10.1109/BigDataCongress.2015.91

[11] Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David E. Culler. 2013. BOSS: Building operating system services. In *NSDI*, Vol. 13. 443–458.

[12] Suhrid Deshmukh, Leon Glicksman, and Leslie Norford. 2018. Case study results: Fault detection in air-handling units in buildings. *Advances in Building Energy Research* 12, 1 (2018), 1–17. DOI: https://doi.org/10.1080/17512549.2018.1545143

[13] EIA. 2016. Commercial buildings energy consumption survey (CBECS) user's guide to the 2012 CBECS public use microdata file. August (2016). https://www.eia.gov/consumption/commercial/data/2012/index.cfm?view=microdata.

[14] Gabe Fierro and David E. Culler. 2017. Design and analysis of a query processor for brick. *Proceedings of the 4th ACM International Conference on Systems for Energy-Efficient Built Environments* 1, 1 (2017), 11:1–11:10. DOI: https://doi.org/10.1145/3137133.3137155

[15] Gabe Fierro, Marco Pritoni, Moustafa AbdelBaky, Paul Raftery, Therese Peffer, Greg Thomson, and David E. Culler. 2018. Mortar: An open testbed for portable building analytics. In *Proceedings of the 5th Conference on Systems for Built Environments*. ACM, 172–181.

[16] Nelson Fumo and M. A. Rafe Biswas. 2015. Regression analysis for prediction of residential energy consumption. *Renewable and Sustainable Energy Reviews* 47 (2015), 332–343. DOI: https://doi.org/10.1016/j.rser.2015.03.035

[17] Building Re-Tuning Training Guide. [n.d.]. *Occupancy Scheduling: Night and Weekend Temperature Set Back and Supply Fan Cycling During Unoccupied Hours.* Report No. PNNL-SA-85194. Pacific Northwest National Laboratory: Richland, WA. Available at: http://www.pnl.gov/buildingretuning/documents/pnnl_sa_85194.pdf. (Last accessed 10/5/2016.).

[18] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. 2013. SPARQL 1.1 query language. *W3C Recommendation* 21, 10 (2013).

[19] Eamonn Keogh and Shruti Kasetty. 2002. On the need for time series data mining benchmarks. *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'02)*, 102. DOI:https://doi.org/10.1145/775047.775062

[20] Jason Koh, Dhiman Sengupta, Julian McAuley, Rajesh Gupta, Bharathan Balaji, and Yuvraj Agarwal. 2017. Scrabble: Converting unstructured metadata into brick for many buildings. In *Proceedings of the 4th ACM International Conference on Systems for Energy-Efficient Built Environments (BuildSys'17)*. ACM, New York, NY, Article 48, 2 pages. DOI:https://doi.org/10.1145/3137133.3141448

[21] J. Zico Kolter and Matthew J. Johnson. 2011. REDD: A public data set for energy disaggregation research. *SustKDD Workshop* 25, 1 (2011), 1–6. http://users.cis.fiu.edu/lzhen001/activities/KDD2011Program/workshops/WKS10/doc/SustKDD3.pdf.

[22] Pacific Northwest National Laboratory. 2019. Building Re-Tuning Training Guide: AHU Heating and Cooling Control. https://buildingretuning.pnnl.gov/documents/pnnl_sa_88359.pdf.

[23] Yann LeCun, Corinna Cortes, and C. J. Burges. 2010. MNIST handwritten digit database. *AT&T Labs [Online]*. Retrieved from http://yann. lecun. com/exdb/mnist.

[24] Lucid. 2018. BuildingOS. Retrieved from https://lucidconnects.com.

[25] Johanna L. Mathieu, Phillip N. Price, Sila Kiliccote, and Mary Ann Piette. 2011. Quantifying changes in building electricity use, with application to demand response. *IEEE Transactions on Smart Grid* 2, 3 (2011), 507–518.

[26] Clayton Miller and Forrest Meggers. 2017. The building data genome project: An open, public data set from non-residential building electrical meters. *Energy Procedia* 122 (2017), 439–444. DOI:https://doi.org/10.1016/j.egypro.2017.07.400. CISBAT 2017 International Conference on Future Buildings & Districts—Energy Efficiency from Nano to Urban Scale.

[27] Natalie Mims, Steven R. Schiller, Elizabeth Stuart, Lisa Schwartz, Chris Kramer, and Richard Faesy. 2017. Evaluation of U.S. building energy benchmarking and transparency programs: Attributes, impacts, and best practices. Report No. LBNL-2001038. Lawrence Berkeley National Laboratory, DOI:https://doi.org/10.2172/1393621

[28] Andrea Monacchi, Dominik Egarter, Wilfried Elmenreich, Salvatore D'Alessandro, and Andrea M. Tonello. 2014. GREEND: An energy consumption dataset of households in Italy and Austria. In *IEEE International Conference on Smart Grid Communications (SmartGridComm'14)*. IEEE, 511–516.

[29] David Murray, Lina Stankovic, and Vladimir Stankovic. 2017. An electrical load measurements dataset of United Kingdom households from a two-year longitudinal study. *Scientific Data* 4 (2017), 160122.

[30] Song Pan, Xingru Wang, Shen Wei, Chuanqi Xu, Xingxing Zhang, Jingchao Xie, Jess Tindall, and Pieter de Wilde. 2017. Energy waste in buildings due to occupant behaviour. *Energy Procedia* 105 (2017), 2233–2238. DOI:https://doi.org/10.1016/j.egypro.2017.03.636. In *8th International Conference on Applied Energy (ICAE'16)*.

[31] Steve P. Doty PE and C. E. M.2009. Simultaneous heating and cooling—The HVAC Blight. *Energy Engineering* 106, 2 (2009), 42–74. DOI:https://doi.org/10.1080/01998590909509174

[32] Pecan St. 2018. Dataport website. Retrieved from https://dataport.cloud/.

[33] Paul Raftery, Shuyang Li, Baihong Jin, Min Ting, Gwelen Paliaga, and Hwakong Cheng. 2018. Evaluation of a cost-responsive supply air temperature reset strategy in an office building. *Energy and Buildings* 158 (2018), 356–370. DOI:https://doi.org/10.1016/j.enbuild.2017.10.017

[34] Sullivan Royer, Stéphane Thil, Thierry Talbert, and Monique Polit. 2014. Black-box modeling of buildings thermal behavior using system identification. *IFAC Proceedings Volumes* 47, 3 (2014), 10850–10855.

[35] Analytic Rules, Comprehensive Library, Analytic Functions, and Full Programmability. 2012. SkySpark® analytic rules: Combining a comprehensive library of analytic functions with full programmability the tools your need to address your applications.

[36] Jeffrey Schein, Steven T. Bushby, Natascha S. Castro, and John M. House. 2006. A rule-based fault detection method for air handling units. *Energy and Buildings* 38, 12 (2006), 1485–1492. DOI:https://doi.org/10.1016/j.enbuild.2006.04.014

[37] Transaction Processing Performance Council. 2018. TPC-C Benchmark Revision 5.11.0. Retrieved from http://www.tpc.org/tpc_documents_current_versions/current_specifications.asp.

[38] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, Vol. 35. ACM, 230–243.

[39] Thomas Weng, Anthony Nwokafor, and Yuvraj Agarwal. 2013. BuildingDepot 2.0. *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings (BuildSys'13)*. DOI:https://doi.org/10.1145/2528282.2528285

[40] S. E. Womble, J. R. Girman, E. L. Ronca, R. Axelrad, H. S. Brightman, and J. F. Mccarthy. 1995. Developing baseline information on buildings and indoor air quality (BASE'94). Part II (1995), 1–8.

[41] Zheng Yang and Burcin Becerik-Gerber. 2016. How does building occupancy influence energy efficiency of HVAC systems?*Energy Procedia* 88 (2016), 775–780. DOI:https://doi.org/10.1016/j.egypro.2016.06.111 *CUE 2015 - Applied Energy Symposium and Summit 2015: Low Carbon Cities and Urban EnergySystems*.

Received March 2019; revised September 2019; accepted October 2019