David Gamero
George W. Woodruff School of Mechanical Engineering,
Georgia Institute of Technology,
801 Ferst Drive,
Atlanta, GA 30332
e-mail: davidgamero@gatech.edu

Andrew Dugenske
Georgia Tech Manufacturing Institute,
Georgia Institute of Technology,
813 Ferst Drive,
Atlanta, GA 30332
e-mail: dugenske@gatech.edu

Christopher Saldana
George W. Woodruff School of Mechanical Engineering,
Georgia Institute of Technology,
801 Ferst Drive,
Atlanta, GA 30332
e-mail: christopher.saldana@me.gatech.edu

Thomas Kurfess
George W. Woodruff School of Mechanical Engineering,
Georgia Institute of Technology,
801 Ferst Drive,
Atlanta, GA 30332
e-mail: kurfess@gatech.edu

Katherine Fu[1]
George W. Woodruff School of Mechanical Engineering,
Georgia Institute of Technology,
801 Ferst Drive,
Atlanta, GA 30332
e-mail: kfu26@wisc.edu

Scalability Testing Approach for Internet of Things for Manufacturing SQL and NoSQL Database Latency and Throughput

The proliferation of low-cost sensors and industrial data solutions has continued to push the frontier of manufacturing technology. Machine learning and other advanced statistical techniques stand to provide tremendous advantages in production capabilities, optimization, monitoring, and efficiency. The tremendous volume of data gathered continues to grow, and the methods for storing the data are critical underpinnings for advancing manufacturing technology. This work aims to investigate the ramifications and design tradeoffs within a decoupled architecture of two prominent database management systems (DBMS): SQL and NoSQL. A representative comparison is carried out with Amazon Web Services (AWS) DynamoDB and AWS Aurora MySQL. The technologies and accompanying design constraints are investigated, and a side-by-side comparison is carried out through high-fidelity industrial data simulated load tests using metrics from a major US manufacturer. The results support the use of simulated client load testing for comparing the latency of database management systems as a system scales up from the prototype stage into production. As a result of complex query support, MySQL is favored for higher-order insights, while NoSQL can reduce system latency for known access patterns at the expense of integrated query flexibility. By reviewing this work, a manufacturer can observe that the use of high-fidelity load testing can reveal tradeoffs in IoTfM write/ingestion performance in terms of latency that are not observable through prototype-scale testing of commercially available cloud DB solutions. [DOI: 10.1115/1.4055733]

Keywords: IoT, IoTfM, MySQL, AWS, cloud, cybermanufacturing, industrial internet of things, information management

1 Introduction

The internet of things (IoT) brings enhanced productivity to industrial manufacturing environments. The next big leap in manufacturing technology is represented by the German strategic initiative Industry 4.0, in which IoT, Big Data, and Service fundamentally alter production as described by Kagermann and Wahlster [1]. Also referred to as the industrial internet of things (IIoT) or internet of things for manufacturing (IoTfM) [2], this revolution holds the potential to create a tremendous surge in manufacturing productivity, driven by real-time access to vast droves of previously inaccessible data in a granular, non-cost prohibitive format.

The growing intersection of automation and manufacturing has given rise to a proliferation of new sources of information, allowing increasingly sophisticated analysis of industrial data. As the space grows, the methods, machines, and formats of analyzing the data range from low-cost distributed sensors to specialized machine learning compute clusters in the cloud. The backbone of this revolution is access to data that were previously cost-prohibitive to acquire. Declining data storage costs allow historic records to be easily archived for future analysis, and low latency IoT services can provide crucial glimpses into live machine states across the world.

In this emerging space, the need to understand data storage technologies and their tradeoffs is critical to every process that operates on that data. Generalized performance metrics allow comparisons between database management systems (DBMS). As data read and write access are a fundamental activity in digital manufacturing, architectural decisions made, such as the selection of a DBMS, have cascading effects on the performance, scalability, and design constraints for entire installations of industrial sensor systems. Understanding the implications of different query languages and data storage technologies, along with their relative compatibility with respect to industrial sensor installations in manufacturing settings, will be necessary in order to meet the full potential of the digital industrial revolution. This work aims to characterize architectural and design differences between structured query language (SQL) and NoSQL databases through the lens of manufacturing, with a focus on scaling up from a prototype size (<10 client IoT devices) to a production system (>100 concurrent IoT assets). An implementation comparison is carried out using Amazon Web Services (AWS) and simulated client loads seeded with over a year of historical IoT data from instrumented assets at a major US manufacturing firm. Simulated clients use the mean and standard deviation of message size (measured in bytes) to generate randomized data of the same volume and throughput rate as seen in the manufacturing firm.

The main contributions of this work can be summarized as follows:

(1) Simulated client testing in a decoupled IoTfM architecture reveals bottlenecks that are not inherent to any individual

[1]Corresponding author.

Contributed by the Computers and Information Division of ASME for publication in the Journal of Computing and Information Science in Engineering. Manuscript received December 16, 2021; final manuscript received September 8, 2022; published online October 10, 2022. Assoc. Editor: Christopher McComb.

layer within the architecture and could be undetectable at the prototype scale, yet emerge at production scale. Results show that it can be used for benchmarking vastly different database technologies using a common set of metrics, such as latency and throughput.

(2) The process for extracting and generating simulated IoTfM clients that generate randomized traffic while maintaining mean characteristics from a pre-defined backlog of IoT records is demonstrated to provide a testing environment simulating production workloads compared to directly stress-testing the database layer itself.

(3) An approach for measuring database latency and throughput metrics from a decoupled IoTfM architecture was detailed in this work. Results support that the metrics gathered are able to provide non-obvious insights into performance tradeoffs within the database layer of the decoupled architecture. Isolating the latency within the database layer could be used to inform architectural decisions and evaluate potential alternative database technologies for future research on IoTfM system configuration.

2 Background

Several technologies, protocols, and models are used in the field of cyber physical systems (CPS). This section introduces these layers in successive order as each technology builds upon earlier ones. Additionally, proposed architectures for cyber physical production systems (CPPS) are introduced.

Several database technologies and types are described, and existing work investigating databases in both generalized and IoTfM use cases is summarized. Next, one area of investigation is highlighted, and the research questions for this work are introduced.

At the intersection of cyber systems and physical systems, CPS have become increasingly ubiquitous, driven by plummeting hardware costs and commoditized network access in manufacturing environments. As asserted by Mourtzis et al., the amount of data generated as low-cost sensors proliferate is rapidly growing [3]. The work presented in this thesis is highly relevant to the design and data management of CPS.

Message queuing telemetry transport (MQTT) "is designed as an extremely lightweight publish/subscribe (Pub/Sub) messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth" [4]. MQTT makes use of a broker, acting as a message bus, to transfer messages from publishing clients to receiving clients by topic. Topics are arbitrarily designated within an installation to segment information and allow finer subscription granularity. MQTT has found wide use in IoT, as it allows for flexible architectures with multiple clients and subscribers without heavy computational requirements or overhead [5,6].

Alongside hypertext transfer protocol and MQTT, protocols such as constrained application protocol [7] and advanced message queuing protocol [8] are used in the IoT space. For IoT systems, each of these protocols has advantages in compatibility, message size, architecture, and encryption. MQTT is the default protocol for the major cloud providers, including AWS, Google Cloud Platform (GCP), and Microsoft Azure [9].

MQTT supports the transmission of messages with three quality of service (QoS) levels. QoS level 0 does not include message receipt acknowledgment, and messages are only transmitted once. This QoS level has the least overhead, and the tradeoff is that messages are not guaranteed to be delivered. QoS level 1 guarantees a message is received at least once but could be delivered multiple times. A copy is kept by the sender until the message receipt is confirmed. QoS level 2 guarantees a message is received exactly once and includes coordinated transmission in which both ends acknowledge transmission and receipt of the message [4]. However, none of the levels guarantee the order in which messages are transmitted.

The growth of cloud computing is exemplified by a "shift in the geography of computing", in which software is executed on remote computers in data centers accessed through the internet [10]. The tasks involved in managing hardware are handled by cloud providers who offer incremental pricing for access to a set of centralized servers. As defined by The National Institute of Standards and Technology, the cloud computing model allows various clients to share computing resources as services. The main tenet is the ease with which clients can easily change service requirements in a low-cost way [11]. The basic services provided include software, platform, and infrastructure, provided in user-centric and task-centric ways [12]. Major providers include AWS [13], Microsoft Azure [14], and GCP [15], with each offering a variety of services [16].

Data can be transmitted to the cloud from sensors installed on assets. Data can also be transmitted locally or used on-premises. In the decoupled architecture proposed by Nguyen and Dugenske [2], gateways have the potential to aggregate data into a single connection from multiple discrete sensors. The primary purpose of gateways is to convert the specific protocol of the device to the protocol of the architecture. The sensor data are acquired using low-cost hardware and transmitted through the architecture to a Pub/Sub message bus where data consumers can receive it. One of the consumers proposed is an archiving service that inserts data into a database to keep historical records. More distributed proposed architectures include a network connection for each individual sensor.

Bonci et al. [17] proposed an architecture that includes running lightweight SQL databases on each sensor to further decentralize the data. This approach creates duplicate records across the databases and heavily relies on the execution of stored procedures to transmit the new data to synchronize all the independent databases. This work recognized the potential of MQTT as data distribution and Pub/Sub for the distribution of IoT payloads.

The prevalence of IoT has led to the development of standardized offerings from major cloud providers that leverage MQTT to create managed IoT deployments and integrations as a service, known as IoT as a service (IoTaaS). AWS IoT core is a software as a service (SaaS), which can be integrated with infrastructure as a service (IaaS) and platform as a service (PaaS) offerings from Amazon Web Services. Similar to offerings from Azure IoT and Google cloud IoT, each offers Pub/Sub using MQTT with different restrictions including topic names. Client connections in AWS IoT Core are capped in data message transfer frequency by AWS; for example, a single AWS IoT core client is limited to 100 Hz message frequency at the time of writing. Messages that exceed the client message publish quota are discarded. This limits the number of shared connections on a single gateway; however, additional connections are trivial to implement, bypassing this limitation.

Within the 5Cs architecture model proposed and discussed by Lee et al. [18] and Monostori [19], every level is built upon increasingly complex and auto-correlated analysis of underlying sensor data. Beginning at the base level in which a sensor network establishes communication, the data conversion, cyber, cognition, and configuration levels delineate progressively removed and abstract insights.

The architecture is built in an unopinionated way toward data storage but is influenced by latency and throughput limitations of all underlying technologies. As we rise through the layers, the complexity of analysis and the processing power necessary to derive insights increases. Each layer performs additional transformations and calculations on the computed outputs of the layers below it, culminating in layer 5, in which machines can self-configure, self-adjust, and self-optimize.

Data from industrial IoT can grow to volume, velocity, and variety consistent with Big Data [20]. For analyzing industrial Big Data, the same techniques that non-industrial Big Data utilizes are applicable. The use of data lakes to store a combination of structured and unstructured data for archival analysis and insights may lead to the use of cluster-based scalable solutions, such as MapReduce, Hadoop, and Apache Spark. While ideal for large workloads, they require specialized implementations [21,22]. The prevailing

architectural models for CPPS are to store the data in databases or data stores and use queries to extract insights from each layer of data to produce higher-order information [23]. Some proposed architectures [17] suggest SQL databases as an option, while others are unopinionated about database technology. Current NoSQL offerings leverage online transaction processing for models such as document stores and key-value stores.

2.1 Databases. SQL databases differ from NoSQL databases in a series of critical ways. SQL databases have a rigidly defined schema, which requires that the data fields be known in advance to configure the database to store records before receiving them [24]. SQL databases can be configured on a single server with optional additional read replicas, or in a sharded cluster configuration. SQL databases support atomic, consistent, isolated, and durable transactions, creating a reliable enterprise data storage system. The storage-optimized nature of SQL enables flexible querying of rigidly defined data schemas that may require complex relationships [25].

AWS relational database service (RDS) offers single-master and multi-master configurations for MySQL compatible Aurora database clusters. In single-master configuration, write throughput scales vertically with hardware, in that upgrading a dedicated MySQL instance can improve performance, but additional servers cannot be added to achieve a similar increase. Single-master clusters have a single write instance and multiple read replicas that increase data availability. Multi-master configurations allow continuous availability via writing across multiple instances in the cluster. In this thesis, an Aurora MySQL single-master cluster with a single instance was used.

NoSQL databases are frequently key-value or document stores in which a primary key and/or sort key are used to retrieve specific records without a rigidly defined table schema. Records can be inserted with arbitrary fields, and data relationships, such as foreign keys, are not strictly enforced. Data fields and relationships do not need to be known in advance when writing records to a NoSQL database. Additionally, NoSQL data are typically less normalized, leading to duplicate records that must be maintained. NoSQL database to support the high read-write speeds, it must be designed with advanced knowledge of the query structures and access patterns that will be used for retrieving records. This structure is well-suited for high-traffic use with known access patterns but becomes unwieldy with poorly defined access patterns.

Rautmare and Bhalerao [26], and Fatima and Wasnik [27] have found expected results for IoT use cases: the single-transaction latency and write speed of NoSQL databases have been found to increase at a lower rate with increasing transaction volume and frequency when compared to SQL databases in IoT applications. The ability to distribute disk writes across an arbitrary number of commodity instances makes NoSQL write performance excel compared to single-master SQL. The distributed data across instances result in rigidly pre-defined access patterns; therefore, the flexibility of queries is dramatically restricted in NoSQL compared to SQL. Both databases have scalable implementations that sacrifice transaction consistency to achieve higher throughput [28].

Current IoT as a service offerings can have zero-code integrations for writing received records on MQTT topics to both SQL and NoSQL databases, with integrated schema-generators for SQL and automatic primary/sort key configuration for NoSQL. Setup and client/device onboarding can be automated as well, with granular permissions and integrations available on major cloud providers.

Amazon Web Services offer IoT Core, an MQTT broker platform for IoT. Messages relayed to IoT Core are secured using a secure sockets layer (SSL)/transport layer security X.509 certificate [29] and are specific to an AWS account" thing" under which the certificate is issued. Architecture norms use a single certificate per connected device to enable granular permissions and certificate authorization via attached AWS identity and access management roles that specify permissions at a device level. Permissions include topic Pub/Sub access and authorization.

Other tangential data pipeline management systems are data streaming services, such as Apache Kafka [30,31] and AWS Kinesis. These platforms are designed for large-scale event streams and facilitate live analytics. Data streaming services function as an additional data routing layer that regularly includes a destination, such as a database management system; data streaming services act as a buffer and data conduit that aggregates data at a high rate while allowing transformations to be performed live on the data. Throughput and latency in these systems exhibit promising results in IoT applications, such as traffic monitoring [32]. Since they are not databases, they are not examined further in this work.

In summary, work done in the CPS field is inextricably linked with data storage and transmission technology. As the field grows, the volume of industrial data swells, which will create new challenges for data processing scale. Generalized work in distributed computing and database throughput has been extensively applied to manufacturing, but work leveraging the narrower subset of constraints for data architectures in manufacturing settings is limited. Proposed IoT data architectures are diverse and numerous yet manufacturing data processes can be deeply entrenched in momentum as they grow beyond the proof of concept stage.

This work aims to address these needs with the following research objectives:

(1) Characterizing the scalability behaviors of NoSQL and SQL databases in the context of existing CPS and CPPS frameworks as measured by latency.
(2) Enumerating scaling factors and bottlenecks encountered during synthetic load tests seeded with data from a major US manufacturing firm.

In order to investigate these questions, a test approach is proposed, and evaluated with respect to existing CPS and CPPS frameworks. By extracting metadata parameters and statistics from an authentic manufacturing data set, the accuracy of existing generalized work can be examined with respect to the more rigidly defined use case of manufacturing IoT data.

3 Approach

The methods proposed in this section aim to detail a high-fidelity stress testing architecture and its application in characterizing the scaling behaviors of an IoTfM installation using either a NoSQL or SQL database in AWS. The integration of statistics extracted from an active IoTfM installation is used to seed higher-fidelity simulated clients. First, the data set itself is introduced, followed by the steps by which representative statistics were extracted. These statistics capture distribution and average information for two key metrics in this analysis: latency and throughput. The latency corresponds to the full duration from message transmission over MQTT to the time at which the record is written into a final database. Throughput corresponds to the rate at which messages are transmitted through the system. Depending on the architecture of the IoTfM system, this can include many intermediate steps. In this work, the decoupled architecture proposed by Nguyen and Dugenske [2] will be examined. Next, the process for creating simulated clients for load testing purposes is detailed. The software and test plan configuration are introduced. The simulated clients are used to stress test two different databases, AWS Aurora MySQL and DynamoDB.

3.1 Data. Data were collected from a large US manufacturing firm to determine the characteristics of the cyber physical production systems that had been instrumented and running on an active production floor for over a year. Data were queried from a MySQL table of over 100 million records spanning 57 assets. The MySQL message archive table contains a full history of MQTT payloads and transmission timestamps for 18 months of the assets' activity from February 2019 to October 2020.

The system that collected the data followed the decoupled architecture proposed by Nguyen and Dugenske [2] in which data were

received over MQTT and written to an AWS RDS MySQL instance. Non-production assets were detected via a flag on the associated data and excluded from this analysis, as they were used for testing the system and are not representative of an active industrial IoT data system. SQL queries were used to exclude test assets by using a "where" clause to exclude data in a subquery.

The mean and standard deviation of the data payload size for each instrumented asset and the overall dataset were calculated to serve as representative samples. The payload size ranged in the order of magnitude from 1×10^2 to 1×10^3 bytes. The mean and standard deviation of the frequency of message transmission for each asset were calculated. The message transmission frequency per asset had an order of 1×10^1–1×10^2 Hz. The mean and standard deviation for the message transmission frequency were also calculated for the entire data set as calculated with arithmetic mean shown detailed in Eq. (1), and sample standard deviation detailed in Eq. (2). This value was used in conjunction with the data payload statistics to create a test plan in Apache JMeter.

$$\frac{1}{n}\sum_{i=1}^{n} x_i \tag{1}$$

$$S = \sqrt{\frac{\sum_{i=1}^{n} (X_i - \bar{X})^2}{(n-1)}} \tag{2}$$

Sample standard deviation S was calculated using values in the sample set X, the sample set mean $\bar{X}$, and the number of samples n.

3.2 Data Set Schema. Full copies of received MQTT messages were archived in a table along with metadata about the message. Columns used in this work and their datatypes are listed in Table 1. For the purposes of this work, latency and throughput are the most relevant metrics. Average throughput can be calculated directly from the dateTimeReceived column through straightforward manipulation of arithmetic mean, as shown in Eq. (3).

$$\frac{1}{t_{\text{end}} - t_{\text{start}}} |X| \tag{3}$$

where t_{start} and t_{end} are sample time bounds, and X is the set of received MQTT messages received between those bounds inclusively, expression (3) is equal to average throughput.

3.3 Simulated Clients. In order to run load tests, client assets were simulated using Apache JMeter and integrating statistics taken from the data set to reproduce high-fidelity load scenarios. Apache JMeter is an open-source load testing and performance-measuring application built using JAVA [33]. The software runs on the Java virtual machine and supports plugins for interfacing through various protocols including MQTT.

Apache JMeter version 5.3 was used to simulate clients in an AWS cloud environment, transmitting messages from an elastic cloud compute (EC2) instance to the AWS IoT core service endpoint, as depicted in Fig. 1. One client refers to a single sensor, instrument, or data stream source attached to a manufacturing asset. All experiments were run in the US-East-1 region. JMeter was executed from an AWS EC2 instance running the Amazon machine image *amzn2−ami−hvm−2.0.20200904.0−x86_64−gp2sizedasat3.medium*.

Table 1 Manufacturing data set messages schema

Column	Type	Note
Id	int(11)	autoincremented unique identifier
dateTimeReceived	timestamp	UNIX timestamp
topic	varchar	Slash-delimited MQTT Topic
payload	varchar	Stringified Payload JSON object
assetId	varchar	Asset Unique Identifier

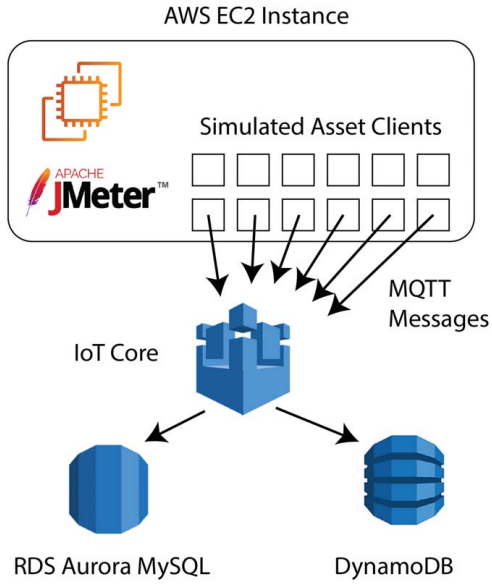

Fig. 1 The virtual test bench leverages simulated asset clients in Apache JMeter running on an EC2 instance to create high-fidelity loading conditions for AWS IoT Core. IoT Core relays messages to DynamoDB and RDS Aurora MySQL database management systems.

The MQTT-JMeter plugin from XMeter-net was used to enable MQTT capabilities within the JMeter stress testing tool. A test plan file was created using the graphical user interface mode, and then executed in command line mode using flags and writing outputs to log files to ensure optimal testing performance.

3.3.1 Test Plan. The JMeter test plan was structured with a thread group containing three main stages: the MQTT connect, message loop logic controller, and MQTT disconnect. An aggregate report listener and summary report listener were used to collect results after the thread group during test plan development. Within the message loop logic controller, a constant throughput timer was used to generate traffic, and the timer was configured using parameters extracted from the data set. A Gaussian timer was added to introduce noise and configured using the parameters extracted from the data set as well.

Within the JMeter test plan, several fields use command line parameter substitution to allow the test plan variables to be modified by passing parameters as flags. This is used later in bash scripting for sweeping the variable space and automating the execution of tests with varied initial conditions. For example: "${P(clients,1)}" substitutes the value of the "clients" parameter from the command line flag, which is passed to JMeter after the flag of the same name prefixed by the letter "J". In the following command, the number of clients is set to ten, which is parsed when the command is run, and substituted into the test plan for each instance of the parameter retrieval.

$$\$./\text{JMeter} - \text{n testplan} \cdot \text{jmx} - \text{Jclients} = 10$$

The test plan was executed via the command line, with two primary parameters passed as flags, *clients*, and *duration*.

Within Apache JMeter, a test plan was created with a single thread group. The thread group was configured with attributes detailed in Tables 2 and 3. The thread group was the singular highest level component in the test plan hierarchy, and it contained the MQTT connect sampler, runtime controller, and MQTT disconnect sampler.

Table 2 Thread group configuration

Setting	Value	Type
Action to be taken after a sampler error	Continue	radio
Name	<arbitrary>	text
Comments	<arbitrary>	text

Table 3 Thread group properties

Setting	Value	Input type
Number of threads (users)	${_P(clients,1)}	Radio
Ramp-up period (s)	0	Number
Loop count	1	Number
Infinite (loop count)	False	Radio
Same user on each iteration	True	Radio
Delay thread creation until needed	False	Radio
Specify thread lifetime	False	Radio

The MQTT connect sampler was added to the test plan as the first child element to the thread group. This element was executed first within each thread in the test plan. It was configured for compatibility with AWS IoT Core. The certificate .p12 file that is supplied was downloaded from IoT Core after issuing a new certificate through the "Add a new Thing" onboarding panel in AWS IoT. The file's contents are not reproduced in this work as it is arbitrary with respect to the methods and results as long as the file is configured as detailed in this work. The certificate was granted read and write permissions for an arbitrarily named MQTT topic that was pre-specified for the duration of all load testing trials. The MQTT connect sampler was configured as detailed in Table 4.

The runtime controller contained the main execution loop. The runtime controller contained elements to carry out two functions: transmitting the MQTT messages and controlling the timing. The runtime controller itself limited the total runtime of each trial and used the duration flag parameter to set the duration in seconds for each trial.

The MQTT pub (publish) sampler transmitted the MQTT messages to the pre-defined topic for each simulated client. The payload was a string in JAVASCRIPT object notation (JSON) format. The JSON fields in the MQTT pub sampler payload are enumerated in Table 5. Configuration parameters for the sampler itself are detailed in Table 6. The time() method was used to retrieve the UNIX time stamp at transmission and embed it in the message, which was parsed later to measure latency through the entire system.

A Gaussian random timer element was configured to introduce noise present in the data sample. The sample standard deviation and mean were configured in this element. The *ConstantDelayOffset* property was set to the mean, and the *Deviation* property was set to the sample standard deviation.

Table 4 MQTT connect sampler properties

Setting	Value	Input type
MQTT connection	AWS IoT Core end-point address	Radio
Port number	8883	Number
MQTT version	3.1	Dropdown
Timeout (s)	10	Number
Protocols	SSL	Dropdown
Dual SSL authentication	True	Radio
Client certification (*.p12)	Certificate from AWS IoT Core	File
Secret	<intentionally blank>	Text
User name	<intentionally blank>	Text
Password	<intentionally blank>	Text
ClientId	conn	Text
Add random suffix for CliendId	True	Radio
Keep alive(s)	300	Number
Connect attempt (sic) max	0	Number
Reconnect attempt (sic) max	0	Number
Clean session	True	Text

Table 5 MQTT Pub Sampler JSON fields and values

JSON field	JSON value	Detail
timeTransmitted	${__time()}	Transmission timestamp
thread	${__threadNum}	Thread id within thread group
numClients	${__P(clients,1)}	Clients parameter
trialDuration	${__P(duration,1)}	Duration (seconds) parameter

3.4 MQTT and Database Ingest Pipelines.

Data were transmitted and relayed using IoT Core integrated actions to write data to a MySQL Aurora cluster via a Lambda function in one set of trials, and a DynamoDB table in the second set of trials. The DynamoDB table was set to enumerated values and write capacity units with auto-scaling enabled.

The MySQL Aurora cluster consisted of a single db.r5.large instance. The MySQL Aurora cluster was configured as a single-master cluster with no read replicas.

3.5 MySQL Aurora Database.

The MySQL Aurora Database was one of the two data destinations for MQTT messages ingested via the AWS IoT Core. Since at this time there is not a direct integration with MySQL Aurora in the IoT Core Actions that were triggered when MQTT messages were received, a Lambda function rule action was triggered that utilized a shared pool of MySQL Connections. The Lambda function was invoked using the IoT Core rule action, including the plain text MQTT message, which was parsed into a JSON object made up of key-value pairs.

The database was initialized with a schema detailed in Table 7 that included a single table using the InnoDB engine. The Aurora Cluster was comprised of a single *db.r5.large* instance function as the reader and writer.

3.6 DynamoDB Database.

The DynamoDB Database was the second of the two potential data destinations for the MQTT messages ingested via the AWS IoT Core. DynamoDB features a direct integration with IoT Core, so the DynamoDB write IoT Core rule action was used to relay information. The DynamoDB table was created with a primary key of the asset ID and a sort key comprised of the UNIX timestamp in milliseconds of the message transmission time.

Table 6 MQTT Pub Sampler configuration parameters

Setting	Value	Input Type
Name	<arbitrary>	Text
Comments	<arbitrary>	Text
Quality of service	0	Dropdown
Retained messages	False	Text
Topic name	<arbitrary>	Text
Add timestamp in payload	False	Radio
Payloads	String	Dropdown
Payload	<see 5>	Text

Table 7 Manufacturing data set messages schema

Column	Type	Note
messageId	int(11)	autoincremented primary key
dateInsert	datetime(3)	
payload	varchar(150)	JSON payload as string

3.7 Latency Measurement. Latency metrics for both the NoSQL and MySQL systems were retrieved using the AWS PYTHON application programmer interface (API). The trial start and end times were recorded from the command line via secure shell for the execution of the load tests, and then entered into the decreasing start and decreasing end configuration variables. Since the architecture includes a Lambda execution for the MySQL insertion, but the DynamoDB integration is fully managed, the full latency for the MySQL configuration is measured from the point of rule invocation onward. For MySQL, this is defined by the Lambda execution duration, as the MySQL Insert operation itself occurs as the final code execution within this duration, and it includes the invocation delay that increases overall insertion latency.

4 Results and Discussion

The results of the proposed synthetic load testing methodology are presented and evaluated in this section. The decoupled digital architecture is analyzed at the database insertion stage for both the NoSQL and MySQL configurations. Also, performance is evaluated in terms of database write throughput and insertion latency across database type and volume of connected simulated clients.

First, the end-to-end characteristics of the proposed synthetic load testing are evaluated for convergence to ensure the data throughput is stable. The stable experiment duration is determined across all proposed client test load sizes to isolate the effect of ramp-up in the time series data.

Next, the write performance of both isolated DBMS configurations is evaluated to establish a baseline performance for later benchmarking. Of the 57 assets included in the data set, 33 remain after excluding the testing/non-production assets. The size in bytes of the MQTT payload for the messages from valid assets is calculated. The mean and standard deviation of this signal are used for the simulated test assets. These data are used to form a model test asset that generates a realistic volume and rate of message data based on the real assets observed. Replicas of this model test asset are then created in the cloud to generate the simulated loading conditions.

Next, the results of request throttling via under-allocation of the *writecapacityunit* parameter are presented. These results include consumed write capacity units, throttled request rate, and rate of client message receiving as recorded by AWS IoT Core Publish In Successes.

Next, the results of the database insertion latency for both NoSQL (DynamoDB) and MySQL (Aurora) are presented. The results are shown for both increasing and decreasing client loads in order to account for auto-scaling momentum in which the DynamoDB throughput could be distorted. Verifying results with both increasing and decreasing client load configurations also accounts for Lambda container re-use that can drastically impact latency via cold-start times. A summary of these results is provided at the end of the section.

4.1 End-to-End Characterization. Figures 2–6 show the end-to-end latency of the system from message transmission to message write completion on the *Y*-axis in milliseconds, plotted over a range of trial durations in seconds on the *X*-axis. Error bars reflect one sample standard deviation. The main objective of the end-to-end analysis is to evaluate the viability of trial durations for later experimentation. One objective is to identify trial durations that are too short, as they experience distortion of the latency by client initialization delays, which are not present in the system at a steady-state. The rapid decline and stabilization with increasing trial duration shown in Figs. 2, 3, and 5 indicate that trials converge on the order of 10^1–10^2 s of trial duration. Figure 7 fails to converge, indicating that the 500 clients' results are not indicative of a steady-state.

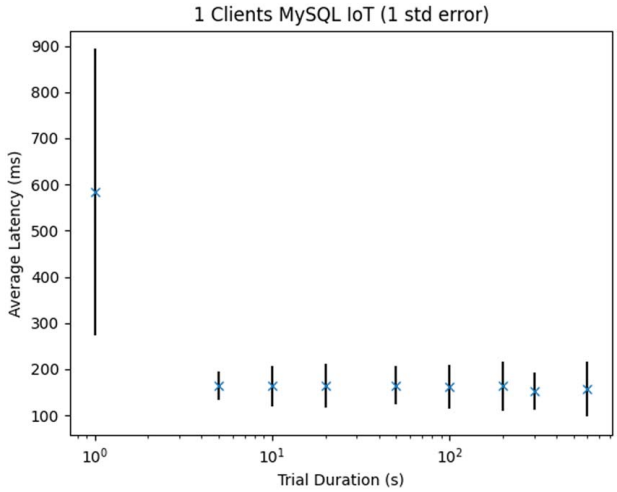

Fig. 2 MySQL average latency with 1 client, variable duration. This figure shows the end-to-end latency of the system from message transmission to message write completion on the *Y*-axis in milliseconds, plotted over a range of trial durations in seconds on the *X*-axis. Error bars reflect one sample standard deviation.

4.2 DynamoDB Writing. After evaluating the end-to-end system with a MySQL configuration, an isolated database verification for auto-scaling and request throttling was executed. Figure 8 shows DynamoDB consumed write capacity units, DynamoDB throttled requests, and IoT Core Successful publishes over MQTT for two trials with DynamoDB configured with 5 and 200 write capacity units, shown on the left and right halves, respectively.

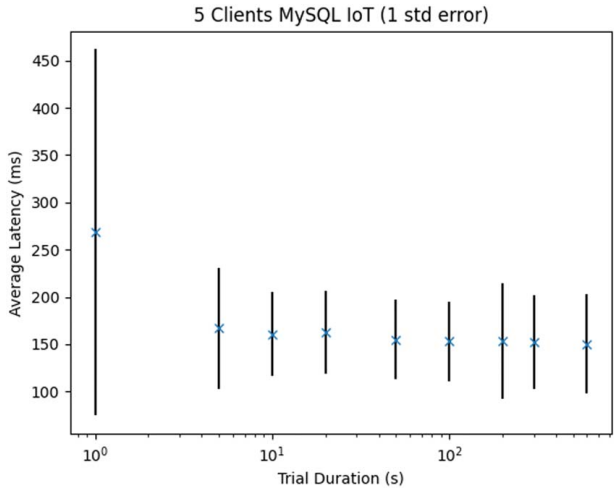

Fig. 3 MySQL average latency with 5 clients, variable duration. This figure shows the end-to-end latency of the system from message transmission to message write completion on the *Y*-axis in milliseconds, plotted over a range of trial durations in seconds on the *X*-axis. Error bars reflect one sample standard deviation.

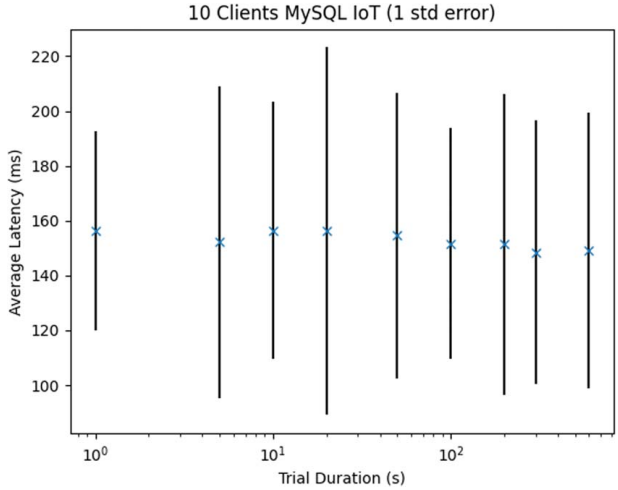

Fig. 4 MySQL average latency with 10 clients, variable duration. This figure shows the end-to-end latency of the system from message transmission to message write completion on the *Y*-axis in milliseconds, plotted over a range of trial durations in seconds on the *X*-axis. Error bars reflect one sample standard deviation.

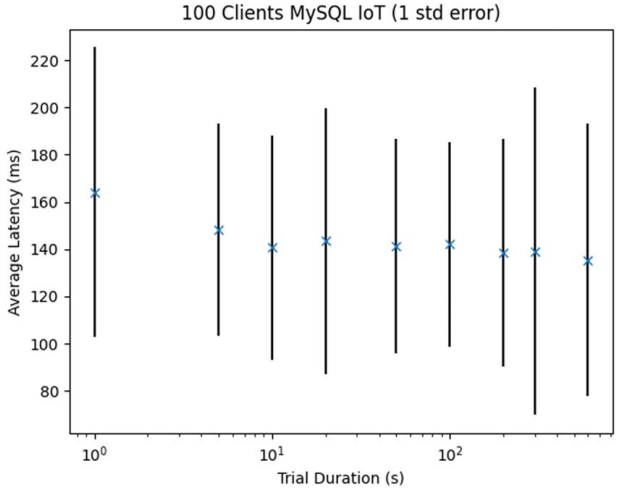

Fig. 6 MySQL average latency with 100 clients, variable duration. This figure shows the end-to-end latency of the system from message transmission to message write completion on the *Y*-axis in milliseconds, plotted over a range of trial durations in seconds on the *X*-axis. Error bars reflect one sample standard deviation.

The simulated client load was 100 clients seeded with statistic parameters from the data set. The throttled requests were exclusively write requests to the DynamoDB table, since no read requests were executed during this time frame. The auto-scaling burst capability of DynamoDB is responsible for the large spike in consumed write capacity units at 19:35 for the 5 write unit trial. The write requests were executed with a latency of less than 25 ms for all messages. With auto-scaling enabled, the 200 write unit capacity was able to service the full load without throttling any requests, while the 5 write units were unable to process the requests and bottlenecked into a large number of throttled requests.

4.3 MySQL Aurora Writing. The same test was executed using the MySQL relational database management system (RDBMS) and observed to meet a write request latency of below 25 ms as shown in Figure 9. These results are consistent with both existing literature and expected performance benchmarks. The MySQL trials used the same simulated 100-client configuration as the NoSQL trial. Throttling conditions were detected via the concurrent Lambda limits since a direct integration with the AWS RDS service for writing records was not available, and Lambda was used to write messages from IoT Core to the RDS Instance. Through vertical scaling via increasing configured instance size, the RDS instance can reach 200,000 writes per second, while DynamoDB by default is limited to 10,000 writes per second per table. The DynamoDB limit can be raised easily to far exceed RDS write limits, but at the cost of losing support for stream-enabled analytics.

4.4 NoSQL and MySQL Load Testing. Given the results of the two earlier sections, the write throughput was serviced fully, resulting in identical throughputs for all configurations from NoSQL and MySQL; however, the latency differed in a statistically

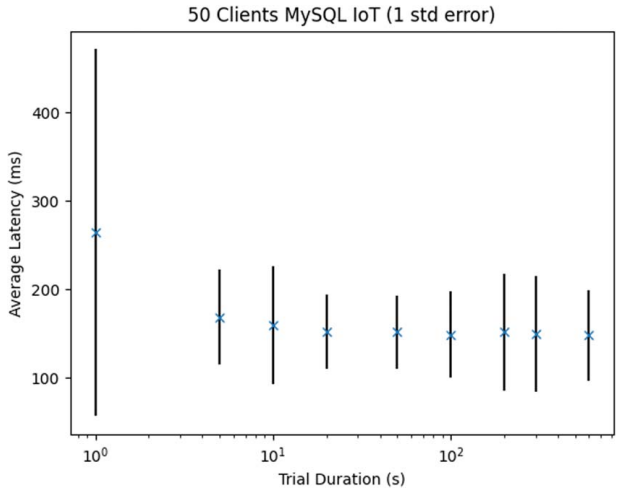

Fig. 5 MySQL average latency with 50 clients, variable duration. This figure shows the end-to-end latency of the system from message transmission to message write completion on the *Y*-axis in milliseconds, plotted over a range of trial durations in seconds on the *X*-axis. Error bars reflect one sample standard deviation.

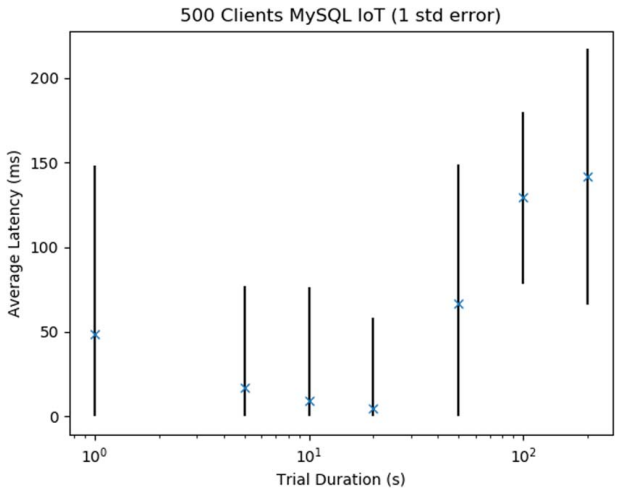

Fig. 7 MySQL average latency with 500 clients, variable duration. This figure shows the end-to-end latency of the system from message transmission to message write completion on the *Y*-axis in milliseconds, plotted over a range of trial durations in seconds on the *X*-axis. Error bars reflect one sample standard deviation.

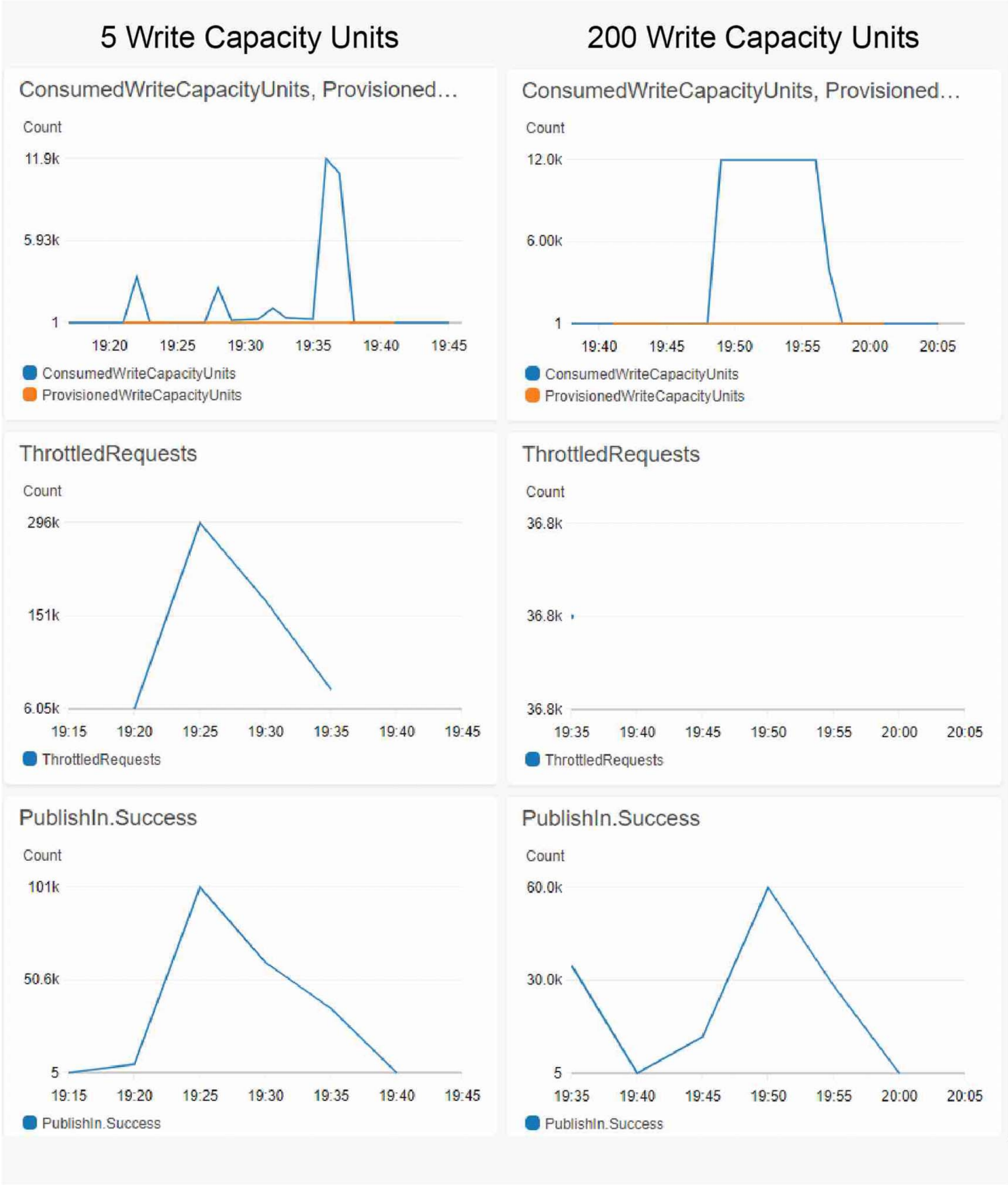

Fig. 8 DynamoDB 100 clients trial. This figure shows event counts on the *Y*-axis in per minute buckets and the *X*-axis reflects time.

significant way. Figure 10 shows the latency of the isolated database portion of both configurations. For the DynamoDB configuration, the insertion is tracked through the managed monitoring solution integrated with AWS CloudWatch. For the MySQL configuration, the latency is primarily impacted by the Lambda function's invocation and execution time. The MySQL database Insert operation duration is included in this metric, as the function does not complete until the insert operation is complete. An increase in rate of execution of Lambda functions can trigger new container provisioning, introducing cold-start delays. Executing the largest number of clients first and then maintaining a monotonic, decreasing number of client connections concentrate the cold-start times in the time before the first trial. Trials with increasing client loads distribute this cold-start latency throughout the trials instead of aggregating most of the delay at the start of the experiment.

To ensure that the effects observed were not driven by Lambda cold-start behavior and DynamoDB auto-scaling momentum, the experiment was also run in reverse, and the results are shown in Fig. 11. Both the increasing and decreasing load configurations show that the DynamoDB latency begins significantly higher and

Fig. 9 Aurora MySQL 100 clients trial. This figure shows event counts on the *Y*-axis in per minute buckets and the *X*-axis reflects time.

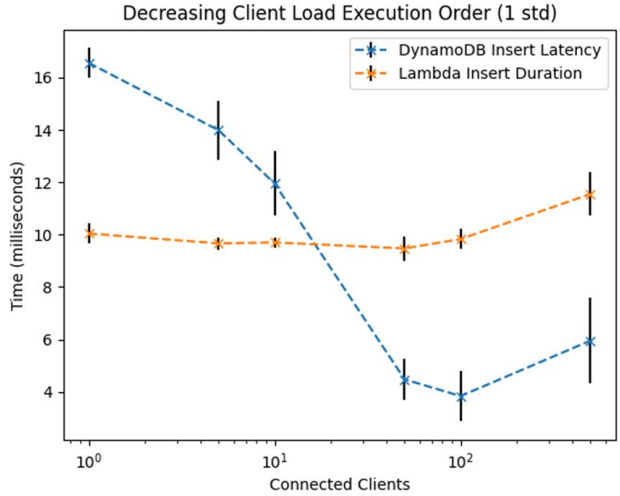

Fig. 10 MySQL and NoSQL isolated latency—decreasing load. This figure shows the end-to-end latency of the system from message transmission to message write completion on the *Y*-axis in milliseconds, plotted over a range of number of connected clients on the *X*-axis. Error bars reflect one sample standard deviation.

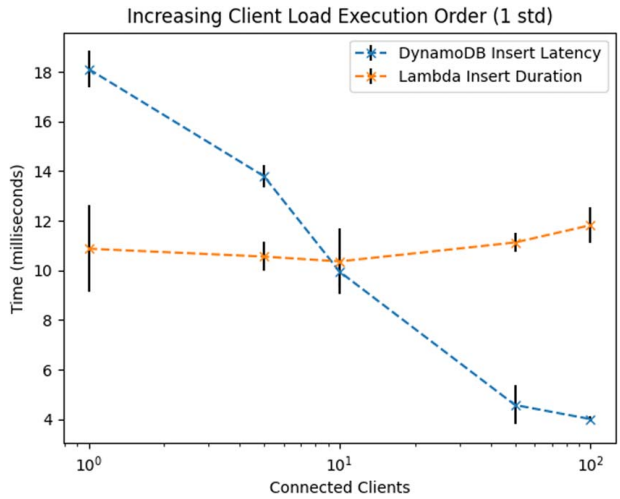

Fig. 11 MySQL and NoSQL isolated latency—increasing load. This figure shows the end-to-end latency of the system from message transmission to message write completion on the *Y*-axis in milliseconds, plotted over a range of number of connected clients on the *X*-axis. Error bars reflect one sample standard deviation.

decreases with an intersection with Lambda insertion latency on the order of 10^1 client connections. The MySQL configuration that used Lambda functions to insert data did not change significantly with respect to the number of clients within the tested range, while the DynamoDB insert latency had an inverse relationship with the number of connected clients. Metrics were retrieved from AWS CloudWatch via the PYTHON API using scripts.

While the performance measurements in this work found an intersection of latency tradeoff between DynamoDB NoSQL and Aurora MySQL, several additional factors are practical in the design of an IoTfM or IIoT system that can have greater impact. By isolating the differences in latency due to choice of database as a relatively small component of the overall latency of an MQTT message's path, the ability to make architectural choices of database technology can be more heavily influenced by other factors, such as price, query flexibility, and ease of integration.

The results support the value of insights provided through client load testing in the prototyping phase, as the latency and throughput respond differently to chosen database technologies as the number of clients increases from the range of 10 to 100. While many systems are tested for a proof of concept using only a small

number of instrumented assets, simulated testing with larger numbers of clients can reveal optimizations that would otherwise not be noticed until dozens of machines were already connected to an implemented system. Using data from already instrumented assets, the results show that the response of database write latency to increase scale can be mapped prior to onboarding larger numbers of assets. The ability to make informed decisions for architecture with respect to latency is especially important for warning systems, where latency can be critical. For small numbers of test clients, warnings would be received quickly, but as the number of clients increases, the choice of optimal database system becomes more complex.

The decoupled architecture used in this work was found to provide effective interoperability between both the DynamoDB and RDS MySQL databases and the IoT traffic ingestion over MQTT. While an integrated IoT hub rule was used to write to DynamoDB, the initial traffic ingestion occurred for both test bench configurations over MQTT. The use of MQTT between loosely coupled components in the data processing pipeline could enhance the applicability of this work across different domains, as MQTT messages are extremely lightweight and can be sent from a variety of sources including other cloud providers, dedicated servers, and embedded devices. The ability to easily swap components within the architecture made isolating the database component significantly more accessible when compared to bespoke, tightly coupled pipelines.

For installations that adopt a decoupled, standardized architecture, the ability to quickly redirect IoT traffic to new destinations could enable faster upgrades and access to a wider range of technologies. Additionally, by using a standardized communication protocol between stages in data processing, multi-cloud configurations could be significantly easier to deploy. Since different cloud providers adjust pricing and deploy new features independently, users of the decoupled architecture could stand to gain more quickly from advances on any cloud provider with potentially drastic reductions in cost of adoption.

The results illustrate comparable performance in terms of latency across the NoSQL and MySQL implementations. As a decoupled architecture was used, the full end-to-end system's latency was found to converge for simulated client loads less than 500 clients, and the convergence was found to occur at these load parameters at trial durations greater than 100 s. The individual performance results from DynamoDB NoSQL confirmed write unit capacity expectations for provisioning throughput, and auto-scaling and burst behaviors were observed to exhibit limited momentum in provisioned throughput. The MySQL Aurora isolated insertion testing was bottlenecked by Lambda executions, as the requirement to trigger Lambda functions from IoT Core rules added an order of magnitude of latency to the system. MySQL Aurora insertions occurred with latency of less than 0.25 ms, while the Lambda invocation and execution introduced greater than 4.0 ms of latency. Both increasing and decreasing client load execution orders of simulated client loading trials indicated that the DynamoDB insertion latency began higher than the Lambda but decreased as client load increased. After an intersection between 10^1 and 10^2 simulated clients, the Lambda configuration maintained its latency, while DynamoDB performed with approximately 50% less latency.

5 Future Research

The work presented here can be expanded in several directions to further enhance and explore the field of IoTfM. Characterizing the impact of instance resources such as random access memory and central processing unit (CPU) core count could provide greater insights into vertical scaling capabilities of these decoupled architectures, revealing optimal points to relieve bottlenecks. This work was limited to only two kinds of databases, both of which were managed AWS offerings. Broadening this work to include both other cloud providers and a greater variety of database technologies would lead to a more holistic picture of databases in IoTfM. Applying the same sampling and simulation methods from this work to other manufacturing firms' historical data sets could provide insights into different industrial data gathering practices that would greatly contribute to the ability to generalize the performance data acquired through these experiments.

5.1 Query Flexibility. As in the practical experiment conducted and described above, the choice of database architecture is not differentiated most significantly by performance when applied to the industrial IoT. The ability to flexibly query data is a feature that is far more developed and advanced on RDBMS systems like MySQL. Flexible queries can be instrumented on top of NoSQL databases via extract transform load pipelines or distributed Big Data approaches, such as Hadoop. For a streamlined architecture in line with the literature, the number of discrete data storage locations is minimized, and systems organically grow from proof of concept and pilot installations. In these scenarios, MySQL's combination of established high vertical scaling write speed limit and facilitation of higher level analytics beyond simple data archiving and arithmetic-based stream metrics allows for faster realization of higher level CPPS systems.

5.2 Latency Sensitivity. Latency from the JMeter EC2 to the IoT Core was negligible throughout the experiments, since the JMeter instance was in the same AWS region as the IoT Core endpoint, and it was connecting within AWS instead of being transmitted from a manufacturing location. This physical co-location is what made it possible to execute the experiments without the introduction of latency noise from outside communications and network traffic that could potentially have vastly greater influence on the metrics measures than the ones investigated in this work. Network and internet service provider offerings could influence the maximum viable throughput from a CPPS when transmitting payloads to the cloud, as would any bottlenecks in factory floor networking. In many manufacturing scenarios, wireless communication over WIFI or Bluetooth is used as well, which introduces another order of magnitude of latency variability. Examining the system-level effects of additional sources of latency such as shared network resources and environment-wide trends in network congestion is another looming problem in need of precise characterization when narrowed to the IoTfM field.

5.3 Architectural Scaling. The high-fidelity simulated industrial MQTT sensor payloads and publish characteristics validate a more specialized testing model to potentially allow higher resolution data on future results compared to generalized DBMS throughput and latency comparisons. Due to the MQTT pub/sub architecture, even for scenarios in which a single MySQL instance would saturate its write capacity, MySQL could be used by splitting data from different assets into separate databases or leveraging multi-master clusters. By horizontally scaling the IoT architecture itself using multiple databases subscribed to distinct topics instead of scaling the database management system, CPPS data throughput bottlenecks can be fully avoided. Fitting a multi-database system in which databases are subscribed to topics over MQTT could benefit from leveraging read replicas to allow analytics to run on separate replicas of the databases without impacting the CPU load of the write instance, maximizing theoretical write throughput.

6 Limitations

6.1 Instance Types. Since results were derived using particular instance sizes and configurations, the results have limited predictive power when other sized instances are used. Memory and CPU limitations on the EC2 instance used to execute the JMeter tests were prohibitive in that it limited tests to below 500 connected clients. While the work aimed to initially explore larger numbers of clients, the simulation framework itself experienced throughput

limitations that necessarily capped the maximum number of clients. Differently optimized instances could present a different set of tradeoffs, such as higher network connectivity speeds or more optimizations toward high thread count computation and parallel processing.

6.2 Uncertainty. While every effort was made to reduce uncertainty in this work, higher numbers of connected clients beyond 500 connected clients contained too much uncertainty to analyze any trends. Isolating trial conditions with respect to trial duration was especially challenging. In this work, only trials in which data points converge to within a single standard deviation were used. The largest source of uncertainty that was eliminated is wireless connection, which can be present in manufacturing settings. The trials were carried out within the same AWS region to reduce latency uncertainty as well.

6.3 Cloud and Database Technologies. Given the vast array of potential database storage technologies, configurations, and hosted services, the work presented here characterizes a particular use case through AWS integrated offerings. Other cloud providers offer different technology stacks and integrations that fundamentally alter both throughput and latency results. Only one platform, AWS, was used to generate the results in this work. Other cloud providers could implement the same technologies in different ways or allow different levels of user control granularity for configuration parameters that were used in this work. Integrated monitoring and metrics interfaces are implemented differently across providers, and methods for gathering the same results outside AWS are not investigated in this work.

6.4 Manufacturing Internet of Things Data Set. The data set used for characterizing manufacturing data used for this work is not a generalized IoTfM data set, and provides a granular, real-world example at the cost of broader scope. The data set is used to ensure fidelity with authentic manufacturing, sensor readings, and conditions. Manufacturing data vary in characteristics, format, and metadata across installations, industries, and environments. The results of this work are necessarily limited to the data that were examined within the manufacturing industry.

6.5 Cloud Services Offerings. In the emerging space of cloud services and hosted databases, services and their availability are regularly subject to change. Future versions of technology offerings can fundamentally alter the functionality and add or remove features. Accounting for long-term trends in feature development is not examined in this work, and all technologies discussed are subject to change.

6.6 Cost and Pricing. Cost and pricing analysis is not included in this report, as it is rapidly evolving in the cloud and IoTaaS space. While theoretical throughput optimizations are promising and enticing, the cost analysis for implementing many of the systems and methods demonstrated in this work can vary by billing model and cloud provider. The sensitivity of private sector implementations to pricing and cost is not examined nor accounted for in this work.

6.7 Environmental Impact. The environmental impact of IoT databases and data ingestion pipelines is not examined in this work. Cloud technology makes it much easier to use a tremendous volume of computational resources, but also can result in a more efficient reuse of servers as idle resources are able to be repurposed by another user. As the volume of data from the IoT grows, the environmental cost of storing such data efficiently can be impacted by the format and system used to store the data. Higher-availability data systems tend to consume more electricity and have larger impacts on the environment. Longer-term storage can reduce responsiveness of data querying but provide potentially dramatic energy savings.

7 Conclusions

The ability to convert ingested data flexibly into higher level insights via dynamic access patterns makes MySQL a strong fit for IoT for manufacturing applications using AWS. Direct write speed and latency at scale yield better performance over 200 k message writes per database per second for NoSQL as compared to SQL, yet the impact could be fully alleviated by splitting data writing across multiple databases using a decoupled architecture with multiple write database instances or multi-master MySQL cluster configurations. The capability to derive complex, dynamic insights from SQL aligns best with Industry 4.0 objectives of smart manufacturing by allowing flexibly defined access patterns, while NoSQL requires well-defined access patterns. Stream and direct storage recall without analytics implementations are better served by the scalability of NoSQL. NoSQL can facilitate lower-level data storage but requires additional technologies to explore higher level insights, and NoSQL can require knowledge of necessary access patterns in advance. The main contributions of this work can be summarized as follows:

(1) Simulated client testing in a decoupled IoTfM architecture reveals bottlenecks that are not inherent to any individual layer within the architecture and could be undetectable at the prototype scale, yet emerge at production scale. Results show that it can be used for benchmarking vastly different database technologies using a common set of metrics, such as latency and throughput.
(2) The process for extracting and generating simulated IoTfM clients that generate randomized traffic while maintaining mean characteristics from a pre-defined backlog of IoT records is demonstrated to provide a testing environment simulating production workloads compared to directly stress-testing the database layer itself.
(3) An approach for measuring database latency and throughput metrics from a decoupled IoTfM architecture was detailed in this work. Results support that the metrics gathered are able to provide non-obvious insights into performance tradeoffs within the database layer of the decoupled architecture. Isolating the latency within the database layer could be used to inform architectural decisions and evaluate potential alternative database technologies for future research on IoTfM system configuration.

7.1 Restatement of Research Questions. This work aims to address these needs with the following research objectives:

(1) Characterizing the scalability behaviors of NoSQL and SQL databases in the context of existing CPS and CPPS frameworks.
(2) Enumerating scaling factors and bottlenecks encountered during synthetic load tests seeded with data from a major US manufacturing firm.

7.2 Answers to Research Questions. The results from this work can be applied to answer the research questions previously enumerated:

(1) The scalability of both NoSQL and SQL databases examined in this work fall within the first two layers of the 5Cs model. The scalability of these systems is critical to enabling higher levels of the model to develop. Within a decoupled architecture, the ability to interchange databases allows for greater flexibility, and the work presented here allows the evaluation of DBMS with respect to performance via latency and throughput analysis. The scalability

of both NoSQL and SQL databases can be compared over increasing client load conditions using simulated clients to determine performance differences. With respect to enabling higher levels of the 5Cs CPS model, MySQL's ability to derive higher level insights and enforce data constraints can offer more towards analytics. NoSQL can be optimized for lower latency in use cases that don't rely on flexible access patterns.

(2) Scaling bottlenecks were encountered both for the synthetic load testing system itself, and with the decoupled architecture model in both database configurations. For the NoSQL DynamoDB configuration, the most prevalent bottleneck observed was the write capacity unit limitation, in which insufficient write capacity units were provisioned, and the throttled insertion requests rapidly grew. For the MySQL configuration, the Lambda function insertion stage was the primary bottleneck, as it introduced cold starts to initialization of the system and with each increase in load. The MySQL configuration also was subject to AWS account limits on maximum concurrent Lambda function invocations; however, this limit can be raised via support tickets. The load testing instance itself experienced a bottleneck in simulated client thread execution for the trials with 500 clients, which could be resolved with vertical scaling via a larger provisioned EC2 instance.

Acknowledgment

This work was supported by the Department of Energy Advanced Manufacturing Office through award DE-EE0008303.

Conflict of Interest

There are no conflicts of interest.

Data Availability Statement

The datasets generated and supporting the findings of this article are obtainable from the corresponding author upon reasonable request.

References

[1] Kagermann, H., and Wahlster, W., 2016, *Industrie 4.0—Germany Market Report and Outlook*, Germany Trade & Invest (GTAI), Berlin, Germany, pp. 1–16.
[2] Nguyen, V., and Dugenske, A., 2018, "An Internet of Things for Manufacturing (IOTFM) Enterprise Software Architecture," Smart Sustain. Manuf. Syst., **2**(2), pp. 177–189.
[3] Mourtzis, D., Vlachou, E., and Milas, N., 2016, "Industrial Big Data as a Result of IoT Adoption in Manufacturing," 5th CIRP Global Web Conference Research and Innovation for Future Production, Patras, Greece, Oct. 4–6, pp. 290–295.
[4] MQTT: The Standard for IoT Messaging, 2020. https://mqtt.org/
[5] Soni, D., and Makwana, A., 2017, "A Survey on MQTT: A Protocol of Internet of Things (IoT)," International Conference On Telecommunication, Power Analysis And Computing Techniques (ICTPACT-2017), Chennai, India, Apr. 6–8, pp. 173–177.
[6] Yokotani, T., and Sasaki, Y., 2016, "Comparison With HTTP and MQTT on Required Network Resources for IoT," 2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC), Bandung, Indonesia, Sept. 13–15, IEEE, pp. 1–6.
[7] Shelby, Z., Hartke, K., and Bormann, C., 2014, "The Constrained Application Protocol (CoAP),", Internet Engineering Task Force (IETF) RFC-7252, Fremont, CA, pp. 1–112.
[8] Vinoski, S., 2006, "Advanced Message Queuing Protocol," IEEE Internet Comput., **10**(6), pp. 87–89.
[9] Naik, N., 2017, "Choice of Effective Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP and HTTP," 2017 IEEE International Systems Engineering Symposium (ISSE), Vienna, Austria, Oct. 11–13, IEEE, pp. 1–7.
[10] Hayes, B., 2008, "Cloud Computing," Commun. ACM, **51**(7), pp. 9–11.
[11] Mell, P., and Grance, T., 2011, "The NIST Definition of Clouding Computing Recommendations National Inst. of Standards and Technology," NIST Spec. Publ., **145**, p. 7.
[12] Miller, M., 2008, *Cloud Computing: Web-Based Applications That Change the Way You Work and Collaborate Online*, 1st ed., Que Publishing, Indianapolis, IN, pp. 1–312.
[13] Amazon Web Services, https://aws.amazon.com/, Amazon Web Services, Inc., Seattle, WA.
[14] Microsoft Azure: Cloud Services, https://azure.microsoft.com/en-us/, Microsoft Corporation, Redmond, WA.
[15] Google Cloud Platform, https://cloud.google.com/, Google LLC, Mountain View, CA.
[16] Prodan, R., and Ostermann, S., 2009, "A Survey and Taxonomy of Infrastructure as a Service and Web Hosting Cloud Providers," 2009 10th IEEE/ACM International Conference on Grid Computing, Banff, Alberta, Canada, Oct. 13–15, IEEE, pp. 17–25.
[17] Bonci, A., Pirani, M., and Longhi, S., 2016, "A Database-Centric Approach for the Modeling, Simulation and Control of Cyber-Physical Systems in the Factory of the Future," IFAC-PapersOnLine, **49**(12), pp. 249–254.
[18] Lee, J., Bagheri, B., and Kao, H.-A., 2015, "A Cyber-Physical Systems Architecture for Industry 4.0-Based Manufacturing Systems," Manuf. Lett., **3**, pp. 18–23.
[19] Monostori, L., 2015, "Cyber-Physical Production Systems: Roots From Manufacturing Science and Technology," Automatisierungstechnik, **63**(10), pp. 766–776.
[20] Snijders, C., Matzat, U., and Reips, U.-D., 2012, "Big Data: Big Gaps of Knowledge in the Field of Internet Science," Int. J. Internet Sci., **7**(1), pp. 1–5.
[21] Plattner, H., 2009, "A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database," Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, SIGMOD '09, Providence, RI, June 29–July 2, pp. 1–2.
[22] Chaudhuri, S., and Dayal, U., 1997, "An Overview of Data Warehousing and OLAP Technology," ACM Sigmod Record, **26**(1), pp. 65–74.
[23] Kim, J. H., 2017, "A Review of Cyber-Physical System Research Relevant to the Emerging It Trends: Industry 4.0, IoT, Big Data, and Cloud Computing," J. Ind. Inf. Integr., **2**(2), p. 175001.
[24] Egenhofer, M. J., 1994, "Spatial SQL: A Query and Presentation Language," IEEE Trans. Knowl. Data Eng., **6**(1), pp. 86–95.
[25] Stonebraker, M., 2010, "SQL Databases v. NoSQL Databases," Commun. ACM, **53**(4), pp. 10–11.
[26] Rautmare, S., and Bhalerao, D., 2016, "MySQL and NoSQL Database Comparison for IoT Application," 2016 IEEE International Conference on Advances in Computer Applications (ICACA), Tamilnadu, India, Oct. 24, pp. 235–238.
[27] Fatima, H., and Wasnik, K., 2016, "Comparison of SQL, NoSQL and NewSQL Databases for Internet of Things," 2016 IEEE Bombay Section Symposium (IBSS), Baramati, India, Dec. 21–22, pp. 1–6.
[28] Cattell, R., 2011, "Scalable SQL and NoSQL Data Stores," ACM Sigmod Record, **39**(4), pp. 12–27.
[29] Housley, R., Ford, W., Polk, W., and Solo, D., 1999, Internet X. 509 Public Key Infrastructure Certificate and CRL Profile, Technical Report, RFC 2459.
[30] Apache Kafka, Apache Software Foundation, https://kafka.apache.org/
[31] Thein, K. M. M., 2014, "Apache Kafka: Next Generation Distributed Messaging System," Int. J. Sci. Res. Eng. Technol., **3**(47), pp. 9478–9483.
[32] Tärneberg, W., Chandrasekaran, V., and Humphrey, M., 2016, *Experiences Creating a Framework for Smart Traffic Control Using AWS IoT*.
[33] Apache JMeter, https://jmeter.apache.org/, The Apache Software Foundation.