Simulated Annealing Scheduling on Kubernetes Clusters: An E-Health Digital Twin Use-Case

Miltos D. Grammatikakis, George Kornaros, and Mateo Fortea

Hellenic Mediterranean University,
71410 Heraklion, Greece
{mdgramma, kornaros}@cs.hmu.gr, mateofortea.info@gmail.com

Abstract. Scheduling multiple containerized tasks to edge computing nodes requires toolchains that lay out the necessary computational/communication models, algorithms, and methodology. The proposed toolchain performs stochastic optimization using simulated annealing (SA) and aims to support automated pod assignment to edge computing nodes. Different input parameters specify pod computational and communication requirements and edge nodes performance characteristics, as well as mapping options and algorithmic characteristics that affect the quality of the solution. Preliminary results from an eHealth-related digital twin (DT) use-case demonstrate limited load imbalance when scheduling DT workloads consisting of mixed HTTP PUT/POST and GET workloads to worker nodes (Raspberry Pi4) of a Kubernetes edge computing cluster. DT requests support basic create, read, update, and delete (CRUD) operations for modeling virtual real-time replicas of ECG biosensors within a Linux server-based open-source Eclipse Ditto DT management system that integrates the MongoDB database. Finally, we examine database server performance for simple DT-like CRUD operations on a Linux server. Results indicate that LittleD, a lightweight, low-cost, limited SQL-based database, designed initially for small systems, offers a much smaller average latency than MongoDB (used in Eclipse Ditto), InfluxDB, or MariaDB.

Keywords: Container, digital twin, Kubernetes, load balance, pod, simulated annealing.

1 Introduction

IoT applications utilizing cloud resources often suffer from high latency and low network bandwidth. A solution uses hierarchical network operations, shifting computations from the cloud to the edge or fog layers. The solution must readily adapt to manage diverse containerized applications in production environments. Hence, the concept of Kubernetes has evolved, since its inception in the early 2000s, towards lightweight, resource-limited, certified distributions, such as K3S by Rancher Labs; implementations are easy to install, deploy, and manage at the edge [1].

Within this context, the Kubernetes scheduler (*kube-scheduler*) maps pods across available worker nodes in the edge computing cluster. Pods represent a group of one or more containers inside which applications are running; hence, pods are the smallest schedulable units. Scheduling is usually based on the pods' resource constraints and requirements (e.g., CPU and network), and the current resource availability. Hence, the *kube-scheduler* assigns newly created pods to a suitable worker node with at least as

many resources, usually with a fast best-fit algorithm. This algorithm performs two operations in sequence: filtering and scoring. Filtering examines a set of hard constraints that a node must meet to run a pod. Scoring evaluates worker nodes for pod placement against a set of resource-related utilization and load-balancing criteria.

Kubernetes separates worker nodes from control. Worker nodes (k3s agents) support: a) the *kubelet runtime environment* used for ensuring that containers are healthy, and for deploying/managing pods and distributed services, and b) *kube-proxy* for inter-worker communication (via host subnetting), and external communication to the control plane (k3s master). The control plane runs the *API server* for interacting with the cluster, the *kube-scheduler*, the *kube-controller* managers (e.g., for job, node, endpoint, replication, and account services), and a high availability *key-value store* based on *etcd* used for saving cluster state, configuration, and metadata.

Since the default *kube-scheduler* is not aware of pod requirements and the architecture or network topology of worker nodes, it often results in non-optimal pod placement that violates performance requirements. Hence, since 2019, hundreds of research papers have focused on extending the current state-of-the-art in Kubernetes scheduling [2-4]. Furthermore, several practical projects proposing components that make smart Pod placement decisions are supported by the Kubernetes SIG Scheduling Group [5]. Some of these projects contribute to configuration options that control node selection in the scheduler, e.g., by performing placement via label, inter-pod node affinity, and anti-affinity specifications, or using taints/tolerations that provide limits for node sets and pods. Certain pod-related options might also be useful to express, in an abstract way, data locality, inter-worker node workload interferences, and deadlines.

In addition to configuration options, the *kube-scheduler* provides extensions. They extend from calling an external process via HTTPS to executing custom filtering and scoring to enabling multiple plugins per pod that operate on the pod queue or implementing custom scheduling components (or related logic) per pod, usually by directly modifying the default scheduler's source code. Recent extensions of the Kubernetes scheduler (or algorithms tested in simulators) refer to generic scheduling, multi-objective optimization, AI-based scheduling, and autoscaling-enabled scheduling for edge-cloud, fog, or cloud-based configurations [2-4]. However, very few state-of-the-art solutions refer to edge computing, where resources are severely limited, and different challenges apply, such as criticality (soft or hard real-time), performance, scalability, energy efficiency, and reliability constraints.

On the theoretical side, Oleghe proposes heuristic scheduling based on *multi-objective optimization and graph network modeling* concepts for container placement and migration in edge servers [6]. Wojciechowski et al. use *dynamic network measurements* gathered automatically by a dedicated infrastructure layer (implemented via the Istio Service Mesh) to improve inter-application node bandwidth and response characteristics [7]. This approach is crucial for the adoption of Kubernetes in 5G networks. Ogbuachi et al. utilize *real-time monitoring from node resources (CPU, memory, and network load)* to improve the performance and fault tolerance of dynamic orchestration in edge computing applications [8]. Li et al. extend this work by dynamically managing the *disk I/O load* of worker nodes [9] using two policies. Balanced-Disk-IO-Priority (BDI) focuses on I/O balance across nodes, while Balanced-CPU-Disk-IO-Priority (BCDI) considers CPU and disk I/O load imbalance on a single node. Finally, Haja et al. developed a custom, open-source Kubernetes scheduler

extension that uses both *edge reliability constraints and periodic delay measurements* across different edge nodes [10, 11]; their heuristic is suited for latency-sensitive edge applications. This is the only scheduler extension available in the open community.
In this work, we develop an open-source simulated annealing tool and demonstrate its efficiency on dynamic workloads derived from a Digital Twin e-Health use case related to real-time ECG monitoring from wearable BT biosensors.
Like Ogbuachi's work [8], our contribution focuses on context-aware scheduling in edge computing by considering system/network statistics, e.g., CPU and network metrics. However, unlike Ogbuachi's work, our implementation is open-source and considers inter-task synchronization. Moreover, common optimization methods, such as Newton-Raphson, Simplex, Least-squares, and Multistep (used in default *kube-scheduler* and partly in [8]), suffer from the inability to distinguish a local minimum from a global one since they follow the local gradients; thus, they are useful if the starting solution is sufficiently close to the global minimum. However, simulated annealing escapes a local extreme by occasionally moving against the gradient.
While previous scheduling work uses web server or HPC testbenches, our experimental use case demonstrates the efficiency of our scheduling tool using a digital twin of real-time ECG monitoring using Eclipse Ditto with MongoDB. In this context, our work also examines the effect of using basic CRUD operations on different databases, demonstrating the performance of LittleD (limited SQL) over MongoDB (NoSQL, used with Eclipse Ditto), InfluxDB (Time Series), and MariaDB (SQL). Our LittleD comparisons extend the work of Arnst et al. [12], who compared MongoDB, InfluxDB, and MariaDB on storage/retrieval of highly recurrent IoT sensor data. Based on read-and-write performance, resource consumption, and qualitative software complexity metrics, Arnst et al. endorsed InfluxDB as a more suitable DBMS.
Section 2 discusses the architecture of the software tool. Section 3 focuses on analyzing the effectiveness of the mapping tool. It discusses the experimental framework and the digital twin use case. Finally, Section 4 provides a summary and discusses future work.

2 Simulated Annealing Tool

The pod assignment tool inputs: a) a task graph specifying the pods' structure, including the constituent components, their properties, and their relations, and b) a graph modeling the target cluster. Hence, the simulated annealing tool uses the following input parameters for dynamic workload management and resource allocation.

- The task graph specifies computation and (inter-pod) communication requirements for each pod, using a) the number of pods, b) the expected computation load of each pod, e.g., in cycles, c) an affinity flag per pod (-1 if the pod is not fixed to any node, or positive if the pod is fixed to a given node or groups of nodes), d) a matrix providing the communication and synchronization cost among corresponding pairs of pods, e.g., in bytes.
- The target cluster model considers worker node properties, including a) the number of nodes, b) the node type (indexing a hardware file `arch`), c) the number of communication links per node, and d) a matrix describing the link types for each connected node (also indexing `arch`). The file `arch` provides for each node, the

node bandwidth, e.g., in cycles per second, and communication link properties, including send/receive startup time, and maximum link bandwidth, e.g., in bytes/s.

With N nodes and M pods, the total input data amounts to at least $O(\max\{M^2, N^2\})$ elements (with matrix representation), but it can be much less for symmetric pods or node configurations when using an adjacency-list graph representation.

The algorithm optimizes pod assignment to nodes by first calling function `random_solution()` to randomly assign pods to nodes, considering any type of pod affinity (or anti-affinity) properties. Using this preliminary assignment, the tool predicts the total execution delay (our cost model), which considers each pod's computation and communication/synchronization requirements.

Then, the algorithm calls the function `set_starting_temp()` to compute a starting temperature. It does so by performing ten random moves from ten different initial configurations. The starting temperature is set to ten times the (absolute) average change in cost so that large variations are initially possible.

Then, the algorithm repeats the main annealing loop, until reaching a maximum allowable cost, tool execution time, or maximum number of iterations; another possibility is to compare the actual percent change in cost per minute to the expected percent change. Within this annealing loop, another independent chain loop repeatedly calls function `perturb()` to randomly reassign pods to nodes, rerouting communication traffic based on the new mappings until either the percentage of accepted moves or the percentage of improved moves becomes too low, i.e., below the user-defined thresholds. This random perturbation of elements in the existing solution gives a chance for the algorithm to find a better solution, i.e., with a shorter execution time. Perturbation invokes one of three functions in a random, but consistent manner.

- Function `alter_mapping()` alters the assignment of pods by choosing a pod at random and assigning it to a random node. This process is repeated until satisfying basic constraints, e.g., the maximum number of node hops allowed for a message to travel.
- Function `alter_path()` provides a new set of random communication paths. Paths are not altered if the maximum number of node hops is one, which indicates that the network is fully connected.
- Function `alter_prio_order()` selects two pods (assuming they have the same priority) and interchanges their order.

After calling `perturb()` within an inner chain loop, a) the starting temperature for the next annealing loop iteration is increased if the specified minimum fraction of accepted/improved moves is not obtained, or else b) the current temperature is decreased by a user-defined `temperature_decrement` factor. At the end of the chain loop, the `chain_length` for the next chain run increases by a pre-selected `chain_factor`. The minimal cost function encountered over all runs is retained.

3 Digital Twin Use-Case: Scheduling and Database Considerations

The experimental setup for testing the scheduling tool is based on the concept of Digital Twin (DT) [13] [14][15]. DT provides digital replicas of physical devices, processes, or systems that perform real-time monitoring of sensors, analysis, and decision-making.

DTs are used in IoT and Industry 4.0 for predicting performance and wear-out effects based on historical sensor data. In e-Health, they are used for monitoring or diagnosis. In our use case, we have developed a DT representing an STMicro BGW single-lead biosensor operating at 128 or 256 ECG pulses/s with 12-bit precision. The embedded prototype consists of an open-source server that stores the sensors' DT [16] [17]. The server is a Dell PowerEdge T330 with 32GB RAM and 4 HDD drives, running Ubuntu 22.04.2 LTS and Eclipse Ditto 3.3.0 from [17]. Eclipse Ditto integrates the Mongo-DB database, a non-SQL, document database that stores DT data as a hierarchy of key-value pairs. MongoDB [18] provides a rich query language with a wide range of DT operations, such as create, read, update, delete, and search using either Web Sockets, or HTTP PUT/POST, GET, DELETE, and FIND calls. Connectivity from each BT-based, BGW biosensor to the server requires a BT-to-Ethernet driver running on a Raspberry Pi 4 device connected to a 2.1 Gbit/s TP-Link Archer C5400 router. A typical JSON key-value pair of the BGW DT includes access control, attributes (manufacturer, gateway info), features (BGW details), and properties (ECG array, date/time, and rate).

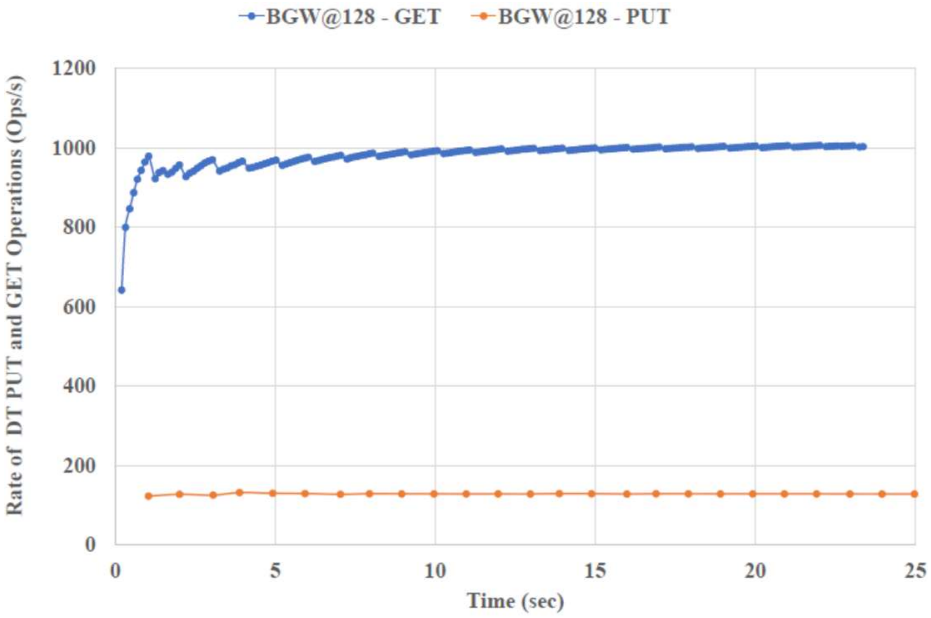

Fig.1 Performance of DT operations: PUT vs. GET.

Experiments on a custom Kubernetes (k3s) cluster with 3 to 5 Raspberry Pi 4 nodes (one K3S server and two to four K3S worker nodes) reveal the relative cost of DT operations. For example, as shown in Fig. 1, GET achieves a much higher rate (up to 8 times more for low rates) than PUT. Hence, using this as a fact, we test the ability of our simulated annealing tool to distribute efficiently a mix of multiple HTTP (GET and PUT) operations that correspond to digital twin transactions across multiple Kubernetes worker nodes resident in a homogeneous Raspberry Pi 4 cluster, before actually deploying them on the cluster.

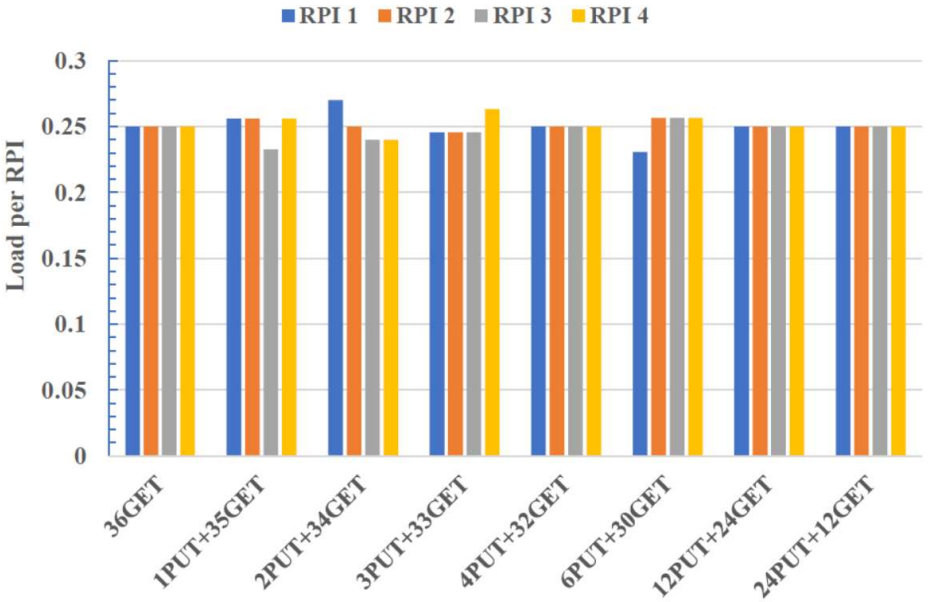

Fig. 2. Load imbalance for simulated annealing with (x PUT and y GET) operations, where (x, y) = (0, 36), (1, 35), (2, 34), (3, 33), (4, 32), (6, 30), (12, 24), and (24, 12).

Fig. 2 shows small load variations when scheduling different collections of 36 GET/PUT operations using simulated annealing on a cluster of four identical worker nodes; we use parameters: `iterations=40`, `chain_length=10`, `chain_factor=1.1`, `temperature_decrement=0.7`, and all move acceptance factors as 0.001. By averaging, for all sets of operations and cluster nodes, the absolute difference from an ideal load of 0.25, we compute an average load imbalance of 1.75% compared to 11.1% for the multistep *kube-scheduler*; multistep fares well only for symmetric cases. Note that, if we were to reduce `iterations=10`, the average load imbalance would worsen to 5.5%, which is still better than multistep, but the tool would be seven times faster (and run in ~10ms). Moreover, if we were to scale up to 72 GET/PUT operations (with a similar distribution) and eight identical nodes, the simulated annealing tool would be fifteen times slower (and run in ~0.7sec), but its load imbalance would remain smaller than multistep: 2.4% vs 5.4%.

Next, focusing on the application, we compare the average latency of common DT-like operations: Create, Read, Update/insert, and Delete (CRUD) on the Ubuntu server. We consider different DBMS under Linux: MongoDB 4.4.6 (NoSQL) [18], MariaDB (SQL) [19], InfluxDB (Time Series) [20], and LittleD (limited SQL) [21]. LittleD was designed primarily for sensor nodes and embedded devices with extremely limited resources [22]; it can perform joins and selections in real-time on 16-bit AVR boards with 16 KB ROM for compiled code, 2 KB RAM, and less than 1 MB of stable storage. In this work, all testbench clients are in C/C++ under Linux [23]. Moreover, in the testbench, 32-bit integers are used (to align with ECG data). The DB sensor record is 1K, and the latency is averaged over 1K iterations, for a total dataset size of 32 MB.

As shown in Fig. 3, LittleD is faster for all CRUD operations. Its average latency is one to two orders of magnitude faster for most operations. The exceptions are: LittleD insert

is only three times faster than MongoDB update, and LittleD read is 5.5 times faster than MariaDB read. LittleD's performance is due to a small parser (limited SQL), query translation while parsing, and use of static memory. LittleD supports create/drop table, read, insert, delete, select, strings, and date/time, but not advanced functions, such as update, join, transactions, triggers, indexes, views, subqueries, and JSON functions.

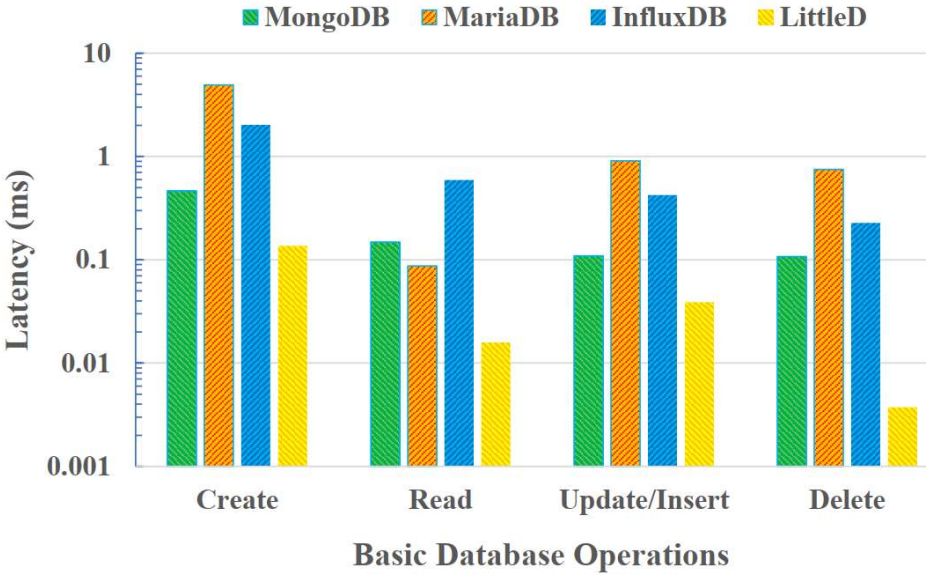

Fig. 3. Latency of DT-like CRUD database operations; since LittleD does not support update, insert (and delete) are provided for comparison.

MongoDB is the second fastest database; observe that for large read workloads (e.g., burst of 1K operations), MongoDB improves fast and outperforms MariaDB. Due to buffering and compression, InfluxDB has low speed (third fastest); however, it also improves fast, especially for bulk reads. LittleD also uses the smallest RAM: ~3MB compared to ~15MB for the other databases. In terms of CPU load, MongoDB exhibits the highest CPU utilization rate for all operations: 28% to 50% compared to 1% to 15% for the other databases; it is even higher for bursts (MongoDB quickly spikes to 100%).

4 Conclusion and Future Work

We have proposed a simulated annealing tool for scheduling workloads on Kubernetes clusters and demonstrated its efficiency on an e-Health digital twin application. The tool can be ported as an external policy of the *kube-scheduler* [11] to map pods to edge computing clusters in a fast and scalable way. It can be extended to perform dynamic pod reassignment. It is also interesting to adapt the tool towards hard real-time and mixed criticality by developing new perturbation policies and cost models.

Another direction is to compare container-based Kubernetes to a VM-based Nginx-HAProxy-Keepalived infrastructure. The latter leverages network technologies to incrementally build a scalable, resilient webserver ecosystem to support HTTP load

balancing and high availability. The solution harnesses Nginx for the webserver, HAProxy as the load balancer, and Keepalived for failover, alike Kubernetes replicas. Finally, concerning the application, it is essential to extend the open-source LittleD database beyond CRUD operations and develop a lightweight, high-performance, low-cost SQL-based Digital Twin ecosystem targeting small Linux-based systems.

Acknowledgments. This work was supported in part by the EU H2020 Project FLUIDOS under GA No. 101070473.

References

1. H. Koziolek and N. Eskandani. “Lightweight Kube distributions: a performance comparison of MicroK8s, k3s, k0s, and Microshift,” *in Proc. Int. Conf. Perf. Engin.,* 2023, pp. 17—29.
2. Senjab, K., Abbas, S., Ahmed, N., et al. “A survey of Kubernetes scheduling algorithms,” *J. Cloud Comput.*, **12**, 87 (2023), pp. 1—26.
3. Rejiba Z., and Chamanara J., "Custom scheduling in Kubernetes: a survey on common problems and solution approaches," *ACM Comput. Surv.*, **55-7, 151**, 2023, pp. 1—37.
4. Carrión C., “Kubernetes scheduling: taxonomy, ongoing issues and challenges,” *ACM Comput. Surv.*, **55(7)**, 2022, Article 138, pp. 1–37
5. Scheduling SIG, https://github.com/kubernetes/community/tree/master/sig-scheduling
6. Oleghe O., “Container placement and migration in edge computing: concept and scheduling models,” *IEEE Access*, **9**, 2021, pp. 68028—68043.
7. Wojciechowski L. K., Opasiak K., and Latusek, J. et al., “NetMARKS: Network metrics-aware Kubernetes scheduler powered by service mesh,” *in Proc. IEEE Conf. Comp. Comm.*, 2021, pp. 1—9.
8. Ogbuachi M.C., Gore C., Reale A., et al., “Context-aware K8S scheduler for real-time distributed 5G edge computing,”, *in Proc. Int. Conf. Softw., Telecom & Comp. Netw.*, 2019, pp. 1—6.
9. Li D., Wei Y., and Zeng B., “A dynamic I/O sensing scheduling scheme in Kubernetes,” *in Proc. ACM Int. Conf. High Perf. Compilation, Comput. and Comm.*, 2020, pp 14–19.
10. Haja D., Szalay M., Sonkoly B., et al., “Sharpening Kubernetes for the Edge,” *in Proc. ACM SIGCOMM Conf.*, 2019, pp. 136–137.
11. Kubernetes Edge Scheduler, https://github.com/davidhaja/kubernetes-edge-scheduler
12. D. Arnst, T. Herpich, V. Plenk, et al., “Comparative evaluation of database read and write performance in an IoT context,” *Int. J. Adv. Internet Tech.*, **12 (1-2**), 2019, pp. 22—38.
13. M.W. Grieves, “Digital twins: past, present, and future. In: N. Crespi, A.T. Drobot, R. Minerva, (eds) *The Digital Twin*, Springer, 2023.
14. L. Li, S. Aslam, A. Wileman, et al., “Digital twin in the aerospace industry: A gentle introduction,” *IEEE Access*, **10**, pp. 9543–9562.
15. L.U. Khan, Z. Han, W. Saad, et al., “Digital twin of wireless systems,” *IEEE Communications Surveys & Tutorials*, 2022, pp. 2230–2254.
16. Eclipse Ditto, Documentation: https://eclipse.dev/ditto, and Software Packages: https://github.com/eclipse-ditto/ditto and https://github.com/eclipse-ditto/dittoexamples
17. S. Ninidakis, M.D. Grammatikakis, G. Kornaros, et al., “Digital Twins for Remote ECG Monitoring,” *in Proc. Int. Conf. Applications in Electronics Pervading Industry, Environment and Society*, Springer, LNEE 1110, 2024, pp. 346—352.
18. MongoDB, https://www.mongodb.com/docs
19. MariaDB, https://mariadb.com/docs
20. InfluxDB, https://docs.influxdata.com
21. LittleD, https://github.com/graemedouglas/LittleD
22. G. Douglas and R. Lawrence, “LittleD: an SQL database for sensor nodes and embedded applications,” *in Proc. Symp. Applied Comput.*, 2014, pp. 827—832.
23. LittleD database comparisons, https://github.com/mfortea/FS_DB_Project