Developing Digital Twins with GridDB

Version 1.0

April 15, 2025

Fixstars Solutions, Inc.

Table of Contents

Introduction

In this document, we will introduce the concept of a Digital Twin and how they are implemented and used before showcasing a sample implementation that monitors, controls, and predicts performance of a water boiler using GridDB, a high performance open-source database with both SQL and NoSQL interfaces and Kafka, an event streaming platform for building data pipelines.

What is a Digital Twin?

A Digital Twin is a virtual representation that emulates an actual product, process or system that is used for monitoring, control, simulation, or testing. Compared to a simulation, the Digital Twin can react in real time to data produced by the actual system.

The digital representation of an actual system can take several forms including a 2D or 3D model to visualize and control the system, a physics-based or data-driven model that simulates the state of a system or a combination that offers both visualization and simulation of the original system.

While it is common to use a twin to compare behavior of an actual product versus its model, in some cases, the twin is developed before the actual system and is used to simulate and test the actual system

Digital Twins can improve operational efficiency and decision making by streamlining monitoring, reducing costs and improving productivity to perform predictive maintenance by utilizing predictive models in the twin and improve communication and customer satisfaction by providing a unified view of the system.

Use Cases

Digital Twins are used in many different industries for many purposes. Some examples include:

- Digital Twins are commonly used in Building Management for Monitoring and Control.
- In design activities for the Automotive, Aerospace and other mechanical engineering fields, Digital Twins are used for prototyping, simulation and testing.
- For users of industrial or other complex equipment and processes, Digital Twins are used for monitoring, control, and predictive maintenance.
- Manufacturers can use Digital Twins to track the production cycle to find inefficiencies or bottlenecks in their processes.

In Monitoring & Control use cases, the Digital Twin offers a virtual representation of the actual system, showing a 2D or 3D model of the system and its state.

When a Digital Twin is used for simulation, environmental and other sensors from the actual device are used either in a physics-based model or a data-driven model that uses machine learning and other techniques that predicts the behavior of the system creating a “Predictive” Twin. A twin based on a physics-based uses fundamental laws and principles of physics to accurately simulate and predict its theoretical behavior. A Data-driven twin, uses previous sensor data along with machine learning or other techniques to build a model to predict the physical system’s behavior.

A Predictive Twin can be used in predictive maintenance and other use cases of Digital Twins to combine monitoring and simulation, using a model based on past and current sensor data to predict when in the future behavior such as when components in the system will fail.

Common Digital Twin Implementations

Many Digital Twin systems are built from scratch using both closed and open source platforms. While it is possible to build a twin system based on a simple message transport system such as MQTT and REST with or without a database, more featureful data streaming platforms such as Kafka and advanced time series databases like GridDB will simplify implementation.

Both AWS and Microsoft offer Digital Twin implementations that are well integrated with their respective ecosystems. Both focus on 3D modelling and monitoring of a physical system along with spatial structures to define hierarchical groups of systems.

Physics-based or data-driven simulations can be developed using Matlab that are integrated with sensors via Mathworks' Simulink product while Autodesk offers their Tandem Digital Twin solution for building design and monitoring.

GridDB and Digital Twins

As a Digital Twin can double the number of writes to the database for a given number of data points, GridDB's high performance ingestion is a key to reducing costs of operating a Digital Twin. With GridDB's Key-Container architecture, data from different sensors are stored in different containers or tables allowing for database access from the Twin to not interfere with data access for the physical systems.

Meanwhile, GridDB's ACID guarantees (Atomicity, Consistency, Isolation, and Durability) ensure that the state between the actual system and the Digital Twin remain correctly synchronized and that the twin is not utilizing a mix of new and historic data for its visualization or simulation.

Finally, GridDB still enables developers to use full SQL statements to build applications instead of just simple NoSQL queries. This is important as SQL join statements that can compare the actual and twin states are straightforward to implement to determine if anomalous behavior is occurring.

Design and Implementation

To showcase developing a Digital Twin stack using GridDB, a simple proof of concept was developed to demonstrate a Digital Twin with monitoring, control and physics-based simulation capabilities.

The temperature of the tank (and outlet flow) and SCFM of natural gas required to heat the boiler are calculated using standard BTU based formulas, if temperature is below the thermostat setting:

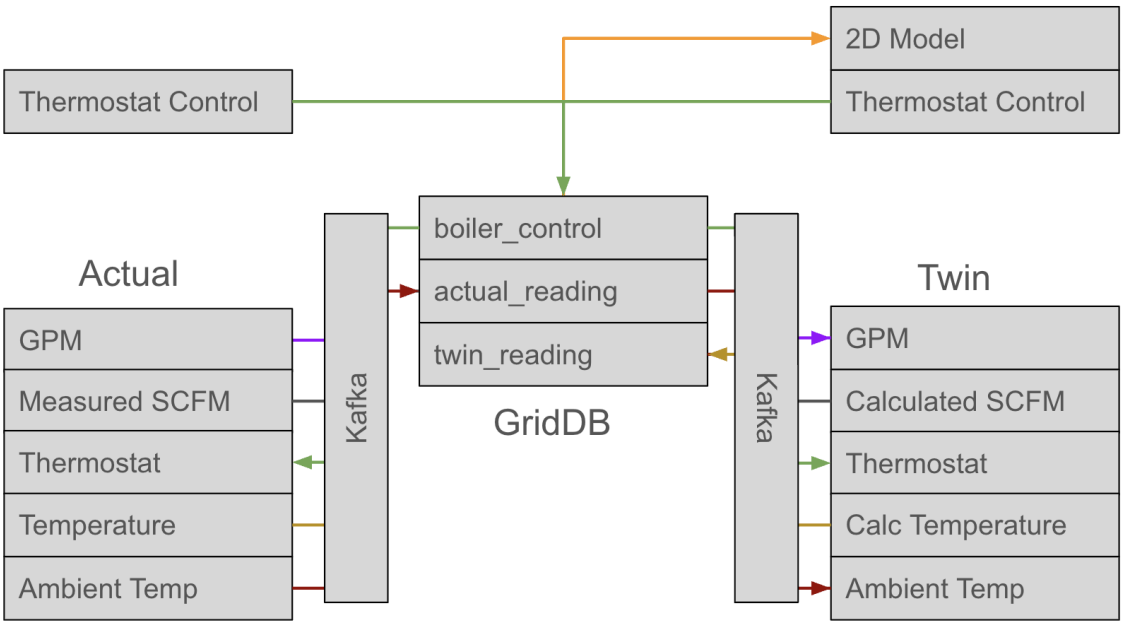
$$temp = temp + (((scfm / 96.7) * 10000)/ (60*tank\_size*8.33))$$
$$scfm = max\_scfm$$

Otherwise we calculate the temperature drop of the tank and the SCFM required to heat it back to the thermostat setting:

$$temp = (gpm*ambient+(gallons-gpm)*temp)/gallons$$
$$scfm = ((8.33 * gpm * (tstat - ambient))/10000)*96.7$$

Data Flow

Kafka is used to stream data between the actual system, the twin and GridDB while the 2D model pulls data continuously from GridDB via the GridDB WebAPI.



The GridDB Kafka Sink plugin is used to push data directly to GridDB from the actual and twin while data is pushed to the twin from GridDB using the Kafka GridDB Source Plugin. Alternatively, the system could be configured to not require the GridDB Source Plugin and the twin would directly read messages sent from the actual system which would offer lower latency but does not guarantee that all messages are synchronized in GridDB.

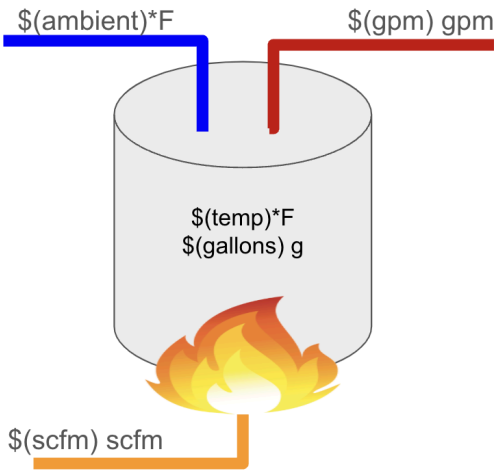
Both the actual and twin read from the same boiler_control Kafka topic to set the thermostat.

Sample Use Case

To demonstrate the usefulness of the Digital Twin concept with GridDB, we will show a proof of concept that is able to visualize the performance of a gas burning boiler while the twin calculates the expected behavior to determine the performance of the actual system.

We start with a simulated gas burning water boiler. The boiler burns a measured amount of natural gas in SCFM to heat a tank where heated water flows out and is replaced by water of ambient temperature. Such systems are commonly used both in manufacturing processes

where hot water is used for a variety of reasons such as cleaning equipment and in building management for radiant heat and hot water consumption.



The thermostat is set via data sent to the Kafka boiler_control_N topic and is read and applied in a secondary thread:

```
def control_thread():
    print("Starting control consumer...")
    consumer = KafkaConsumer('src_boiler_control_1',
            bootstrap_servers=['localhost:9092'])

    global tstat
    for message in consumer:
        data = json.loads(message.value.decode('utf-8'))
        tstat = data['payload']['tstat']
```

Values for ambient temperature and gpm are random and vary between 0 and 10 gallons per minute and 50 and 90 degrees fahrenheit respectively while the value for Gallons is hardcoded. SCFM and temp are calculated every minute with a physics-based model using the following logic:

```
def calc():
    global temp
    global scfm
    global tstat
    global gpm
    global ambient
    global gallons

    gpm = gpm + random.randrange(-5, 10, 1)/2
    if gpm < 0:
        gpm = 0
    elif gpm > 10:
        gpm = 10
```

```
    ambient = ambient + random.randrange(-2, 2, 1)/2
    if ambient < 50:
        ambient = 50
    elif ambient > 90:
        ambient = 90

    if temp < tstat:
        print("Heating tank")
        scfm = max_scfm
        temp = temp + (((scfm/96.7)*10000)/(60*gallons*8.33)*add_error())

    else:
        temp = (gpm*ambient+(gallons-gpm)*temp)/gallons
        if temp > tstat:
            print("Cooling");
            scfm = 0
        else:
            print("heating flow")
            req_btu = 8.33 * gpm * (tstat - ambient)
            scfm = ((req_btu/10000)*96.7)/add_error()
            temp = tstat

    produce()
```

The produce() function then publishes the data via Kafka where it is stored in GridDB and then propagated to the twin.

To simulate anomalous data from an actual device, an error quotient is applied to each value on regular intervals, reducing the temperature or increasing the SCFM.

```
def add_error():
    global error
    global error_count
    error_count = error_count+1
    if error_count == 30:
        error = not error
        error_count = 0

    if error:
        return 0.75
    else:
        return 1.0
```

Data Schema

For each system, there are three time series containers where N is an unique identifier for the system:

- actual_reading_N

- twin_reading_N
- boiler_control_N

actual_reading_N and twin_reading_N have matching schemas and store sensor data while boiler_control_N stores the thermostat setting.

(actual|twin)_reading_N:
- TIMESTAMP ts
- FLOAT temp
- FLOAT scfm
- FLOAT gpm
- FLOAT ambient
- INTEGER gallons

boiler_control_N:
- TIMESTAMP ts
- INTEGER tstat

The Twin

The Twin uses the same thermostat setting control thread and physics-based model as the actual simulated device but instead of generating random data for GPM or Ambient Temperature, it uses the synchronized values from the actual system.

```
def reading_thread():
    print("Starting reading consumer...")
    consumer = KafkaConsumer('src_actual_reading_1',
            bootstrap_servers=['localhost:9092'])

    global temp
    global gpm
    global ambient
    global gallons
    global last_ts
    global scfm

    for message in consumer:
        data = json.loads(message.value.decode('utf-8'))

        gpm = data['payload']['gpm']
        ambient = data['payload']['ambient']
        gallons = data['payload']['gallons']

        if last_ts == None:
            temp = data['payload']['temp']
            last_ts = data['payload']['ts']
        else:
            last_ts = data['payload']['ts']
            if tstat == None:
                print("tstat not set")
            else:
                delta = data['payload']['ts'] - last_ts
                if temp < tstat:
                    print("Heating tank")
                    scfm = max_scfm
                    temp = temp + ( ( scfm / 96.7 ) * 10000)/ (60*gallons*8.33)
```

```
        else:
            temp = (gpm*ambient+(gallons-gpm)*temp)/gallons
            if temp > tstat:
                print("Cooling");
                scfm = 0
            else:
                print("heating flow")
                req_btu = 8.33 * gpm * (tstat - ambient)
                scfm = (req_btu/10000)*96.7
                temp = tstat
        produce()
```

Querying The Twin

As GridDB supports both the ultra fast NoSQL interface and fully featured SQL interface, building queries that leverage the data created by the twin is quite simple.

For the NoSQL interface, the logic is mostly handled by the application. For example, if we wanted to set an anomaly condition when the actual temperature is 10% different than the twin's temperature, the logic would look like:

```
if actual[ts]['temp'] > twin[ts]['temp']*1.1 or
   actual[ts]['temp'] < twin[ts]['temp']/1.1:
       anomaly[ts] = True
else:
       anomaly[ts] = False
```

With SQL, we can use JOINs to combine the actual and twin data for easier analysis of current and historic data.

The following query performs a JOIN on the actual and twin tables and it is visually easy to see when SCFM differs between the twin and actual systems.

```
> select actual_reading_1.ts,
    actual_reading_1.temp as actual_temp, twin_reading_1.temp as twin_temp,
    actual_reading_1.scfm as actual_scfm, twin_reading_1.scfm as twin_scfm
  from twin_reading_1
  join actual_reading_1 on twin_reading_1.ts = actual_reading_1.ts
  order by ts desc;

ts                  | actual_temp | twin_temp | actual_scfm | twin_scfm
--------------------+-------------+-----------+-------------+----------
2025-03-17 03:10:27 |         200 |       200 |      120.42 |     90.32
2025-03-17 03:10:26 |         200 |       200 |      136.94 |     102.7
2025-03-17 03:10:25 |         200 |       200 |       161.1 |    120.83
2025-03-17 03:10:24 |         200 |       200 |      120.83 |    120.83
2025-03-17 03:10:23 |         200 |       200 |      108.74 |    108.74
2025-03-17 03:10:22 |         200 |       200 |      120.83 |    120.83
2025-03-17 03:10:21 |         200 |       200 |      120.83 |    120.83
2025-03-17 03:10:20 |         200 |       200 |       102.7 |     102.7
2025-03-17 03:10:19 |         200 |       200 |       114.4 |     114.4
```

```
2025-03-17 03:10:18 |         200 |       200 |      120.83 |    120.83
```

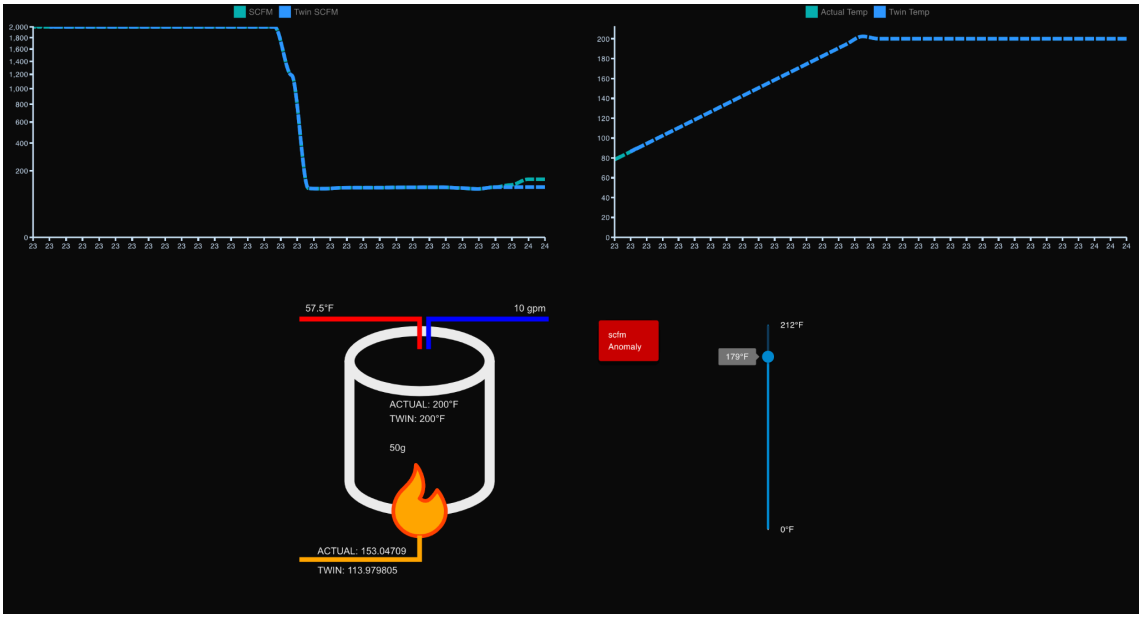
Using aggregation functions, it is easy to count the number of anomalous readings:

```
> select count(*) from twin_reading_1
  join actual_reading_1 on twin_reading_1.ts = actual_reading_1.ts
  where actual_reading_1.scfm > twin_reading_1.scfm*1.1 or
     actual_reading_1.scfm < twin_reading_1.scfm/1.1;

Col1
------
210884
```

Visual Model

The visual model is written in Typescript using Next.js (v15.2.0) as its backbone. Material UI and Tailwind were used for styling; konva was used for drawing SVG lines. Next.js employs a server/client component split; server components are code rendered on the server; client components are rendered by the user's browser.



All GridDB queries are conducted using the GridDB WebAPI on the server side. The WebAPI allows for communication with GridDB using HTTP requests through various HTTP methods ("POST" & "PUT" in this case).

To fetch the current water boiler temperature, for example, a request is made to an API endpoint. This endpoint contains various identifying parameters, such as database and container name. Sent along with that request is a payload body which contains other information which can help narrow the scope of the query.

```
//https://<webapiurl>/griddb/v2/<clusterName>/dbs/<databasename>/containers/<container_name>/rows

async function readContainer (containerName: string, limit: number,
pastHour: boolean) {
    let condition = ""
    if (pastHour) {
      condition = "ts < TIMESTAMPADD(HOUR, NOW(), -1)"
    }
    const raw = JSON.stringify({
      "offset": 0,
      "sort": "ts desc",
      limit,
      condition
    });
    const requestOptions = {
      method: "POST",
      headers: myHeaders,
      redirect: "follow",
      body: raw
    };


    const url = webApiURL + "/containers/" + containerName + "/rows";

    try {
      const resp = await fetch (url, requestOptions)
      const json = await resp.json();
      return json;
    } catch (error) {
      console.log("Error fetching read container: ", error);
    }
  }
```

As with fetching data, we can also write new data to GridDB using the WebAPI. In the case of updating the water boiler temperature, we send a `PUT` request to the WebAPI Server to write a new row of data with the current timestamp and a new value for the temperature.

```
async function pushToBoilerCont (data: any[]) {
    const raw = JSON.stringify([
     //data is an array: [new Date(), sliderValue]
      data
    ]);
    const requestOptions = {
      method: "PUT",
      headers: myHeaders,
      redirect: "follow",
      body: raw
    };

    const url = webApiURL + "/containers/" + BOILER_CONT + "/rows";
```

```
    try {
      const resp = await fetch (url, requestOptions)
      const json = await resp.json();
      console.log(json);
    } catch (error) {
      console.log("Error putting data to " + BOILER_CONT, error);
    }
  }
```

The visual model itself queries GridDB once every second and updates the values on screen accordingly. To go along with this, if there is a noticeable discrepancy between the twin representation of SCFM and the actual SCFM value (>10%), the visual model will display a prominent error message (the temperature is handled the same way).

This is accomplished by comparing the values of the twin and the actual sensor during every query. All values being queried by the visual model are automatically cached by Nextjs on the server side, meaning there are no noticeable performance hits caused by the constant querying and comparing of values.

Conclusion

A Digital Twin is an effective tool to design, evaluate and monitor a physical system by showing a visual representation of the system and predicting or simulating its current and future behavior through physics or data based models.

GridDB's ACID guarantees, high performance, and native NoSQL plus SQL interfaces make it ideal for storing data generated by the actual system and its twin.

In the water boiler use case presented, the Digital Twin provides visualization of the boiler performance and usage as well as using a predictive physics-based twin model that is able to determine when the boiler efficiency is not meeting predicted levels to notify maintenance personnel.

The complete source code for our sample Digital Twin project is available at https://github.com/griddbnet/digital-twin