Technical Report

Title: Difference Between Digital Twin and Simple Simulation

Author: Aditya Adukuri

Date: October 13, 2025

Course/Project: Disaster Response Digital Twin using Open Climate Datasets

# Abstract

This technical report explores the conceptual and practical differences between Digital Twins and Simple Simulations, emphasizing their roles, architectures, and functionalities in the context of disaster response systems. A Digital Twin represents a living, data-connected virtual replica of a physical system, while a simple simulation is a static model used to test predefined scenarios. This report analyzes the two paradigms across dimensions such as data flow, temporal dynamics, fidelity, interaction, scalability, and use-cases. Additionally, the report integrates a practical component with a provided Python implementation that demonstrates basic elements of a digital twin system.

# Table of Contents

1. 1. Introduction
2. 2. Conceptual Overview
3. 3. Architectural Differences
4. 4. Data Integration and Flow
5. 5. Temporal Behavior and Lifecycle
6. 6. Fidelity, Calibration, and Uncertainty
7. 7. Interaction and Control
8. 8. Validation and Trust
9. 9. Scalability and Resource Utilization
10. 10. Comparative Analysis Table
11. 11. Use Cases in Disaster Response
12. 12. Python Implementation Overview
13. 13. Discussion
14. 14. Conclusion
15. 15. References

# 1. Introduction

Digital Twins and simulations both serve as analytical tools for understanding and predicting the behavior of systems. However, their underlying principles, purposes, and implementations differ significantly. The emergence of the Internet of Things (IoT), cloud computing, and artificial intelligence has transformed digital twins into powerful tools for real-time decision-making. This report provides a comprehensive technical comparison of digital twins and simple simulations, focusing on their relevance in disaster response scenarios that utilize open climate datasets.

# 2. Conceptual Overview

A simple simulation is a computational model that represents a real-world process through mathematical equations or algorithms. It operates based on predefined assumptions and inputs, running once or multiple times to generate results. In contrast, a Digital Twin is a dynamic, continuously updated digital representation of a physical entity or system. The key differentiator is the continuous data integration and synchronization that keeps a digital twin 'alive' and reflective of its physical counterpart.

# 3. Architectural Differences

A Digital Twin architecture consists of a physical system, a virtual model, and a data connection layer that synchronizes the two. This tri-layer structure enables real-time monitoring, predictive analysis, and closed-loop control. In contrast, a simulation lacks this data synchronization layer and typically functions as an isolated computational model. Key technologies in digital twins include IoT sensors, cloud data pipelines, AI-based predictive models, and visualization interfaces such as dashboards or GIS systems.

# 4. Data Integration and Flow

Digital Twins depend on real-time or near real-time data feeds from IoT sensors, APIs, or satellites. Simple simulations rely on static or historical datasets. The twin architecture continuously assimilates data, updating parameters dynamically, while simulations must be rerun with updated inputs.

# 5. Temporal Behavior and Lifecycle

Digital Twins operate continuously, maintaining a live state of the physical system. Simulations are episodic—run once, produce results, and terminate.

# 6. Fidelity, Calibration, and Uncertainty

Digital Twins adjust model fidelity dynamically based on data quality and uncertainty metrics. Simulations have fixed fidelity defined at design time.

# 7. Interaction and Control

Digital Twins offer bi-directional interaction—allowing users or automated systems to send control signals back to the real world. Simulations are one-way analytical tools.

# 8. Validation and Trust

Digital Twins undergo continuous validation against real-world observations. Simple simulations are validated offline using historical datasets.

# 9. Scalability and Resource Utilization

Digital Twins require robust infrastructure for streaming data, analytics, and storage, while simulations are compute-efficient for one-time analyses.

# 10. Comparative Analysis Table

The following table summarizes the differences between the two approaches:  
  
Aspect | Digital Twin | Simple Simulation  
-------|---------------|------------------  
Data Flow | Real-time | Static  
Lifecycle | Continuous | Episodic  
Interaction | Two-way | One-way  
Validation | Continuous | Pre-deployment  
Use-case | Operational | Experimental

# 11. Use Cases in Disaster Response

Digital Twins can be used to simulate and manage real-time disaster scenarios, such as floods or wildfires, using live sensor and satellite data. Simulations, while useful for planning, lack adaptive feedback mechanisms necessary for operational response.

# 12. Python Implementation Overview

The attached Python code demonstrates the foundational elements of a digital twin system—data ingestion, state updating, and scenario simulation. It can be extended to include live APIs, ML-driven predictions, and visualization modules.

# 13. Discussion

The integration of real-time data and AI within Digital Twins bridges the gap between static modeling and dynamic decision support. However, developing and maintaining twins requires higher computational and organizational investment.

# 14. Conclusion

Digital Twins extend the concept of simulation by creating a continuously updated, data-driven reflection of reality. While simulations remain essential for theoretical and exploratory analysis, digital twins are the cornerstone for real-time, adaptive system management.

# 15. References

* Tao, F., et al. (2019). Digital Twin in Industry: State-of-the-Art. IEEE Transactions on Industrial Informatics.
* Grieves, M. (2014). Digital Twin: Manufacturing Excellence through Virtual Factory Replication.
* Boschert, S., & Rosen, R. (2016). Digital Twin—The Simulation Aspect. In Mechatronic Futures.
* Batty, M. (2021). Digital Twins and Smart Cities. Environment and Planning B: Urban Analytics and City Science.
* Glaessgen, E., & Stargel, D. (2012). The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles.

# Appendix A: Python Code – digitaltwin.py

"""  
Digital Twin MVP — Flood nowcast prototype (Python script / notebook)  
  
What this does (MVP):  
- Downloads ERA5 hourly precipitation (example) using the CDS API (user must have CDS account and API key).   
- Loads ERA5 into xarray + dask, computes hourly/daily accumulations.  
- Downloads OpenStreetMap data for a bounding box using OSMnx (roads, buildings).  
- Reprojects and writes outputs as Cloud Optimized GeoTIFFs (COGs) for tiled display.  
- Serves a tiny FastAPI that exposes a WMTS-like endpoint for raster tiles and a GeoJSON endpoint for OSM assets.  
- Provides a small Jupyter/Folium client snippet to show a time-slider map of precipitation overlaid on OSM assets.  
  
Notes before running:  
- This is a runnable prototype for local use. You need to install the dependencies listed below and set up the CDS API key.  
- Runtime and data volumes: ERA5 and satellite data can be large. For a quick test, request a small bbox and a short time range (1-3 days).  
  
Dependencies (conda/pip):  
 pip install cdsapi xarray[complete] dask[distributed] rioxarray rasterio rio-cogeo geopandas osmnx folium fastapi uvicorn aiofiles pystac-client  
 # or use conda equivalents  
  
CDS API setup:  
- Create an account at https://cds.climate.copernicus.eu  
- Place your API key in ~/.cdsapirc (example below)  
  
~/.cdsapirc content:  
---------------------------------  
url: https://cds.climate.copernicus.eu/api/v2  
key: <uid>:<api\_key>  
---------------------------------  
  
ALTERNATIVE: If you don't want to download ERA5, skip the CDS parts and use a small sample NetCDF if you have one.  
  
This file is organized as sections. Run in a Jupyter notebook or as a script (some cells require async event loop for FastAPI).  
"""  
  
  
  
  
  
# --- SECTION 0: Imports & config  
import os  
from datetime import datetime, timedelta  
import tempfile  
  
# Data handling  
import xarray as xr  
import dask  
from dask.distributed import Client  
import rioxarray  
import rasterio  
from rasterio.enums import Resampling  
  
# CDS API  
try:  
 import cdsapi  
except Exception:  
 cdsapi = None  
  
# OSM  
import osmnx as ox  
import geopandas as gpd  
  
# Web serving  
from fastapi import FastAPI, HTTPException  
from fastapi.responses import FileResponse, JSONResponse  
import uvicorn  
  
# Map client  
import folium  
from folium import plugins  
  
# rio-cogeo  
from rio\_cogeo.cogeo import cog\_translate  
from rio\_cogeo.profiles import cog\_profiles  
  
  
  
  
  
  
  
  
  
# --- SECTION 1: Dask client (optional but recommended for xarray processing)  
client = Client(processes=False, threads\_per\_worker=4, n\_workers=1, memory\_limit="4GB")  
print("Dask dashboard:", client.dashboard\_link)  
  
  
  
  
  
  
  
  
  
  
# --- SECTION 2: Helper functions  
  
def ensure\_dir(path):  
 os.makedirs(path, exist\_ok=True)  
  
  
def download\_era5\_precip(start\_date, end\_date, area, out\_path):  
 """  
 Download ERA5 hourly total precipitation for the bounding box area.  
 area: [north, west, south, east]  
 out\_path: path to save NetCDF  
 Requires CDSAPI configured (~/.cdsapirc).  
 """  
 if cdsapi is None:  
 raise RuntimeError("cdsapi not installed — install with `pip install cdsapi` or skip this step.")  
  
 c = cdsapi.Client()  
 # construct list of dates and hours  
 dates = []  
 cur = start\_date  
 while cur <= end\_date:  
 dates.append(cur.strftime("%Y-%m-%d"))  
 cur += timedelta(days=1)  
  
 request = {  
 'product\_type': 'reanalysis',  
 'format': 'netcdf',  
 'variable': 'total\_precipitation',  
 'year': sorted(list({d.split('-')[0] for d in dates})),  
 'month': sorted(list({d.split('-')[1] for d in dates})),  
 'day': sorted(list({d.split('-')[2] for d in dates})),  
 'time': [f"{h:02d}:00" for h in range(24)],  
 'area': area, # north, west, south, east  
 }  
 print('Requesting ERA5 with params:', request)  
 c.retrieve('reanalysis-era5-single-levels', request, out\_path)  
 print('Saved ERA5 to', out\_path)  
  
  
def open\_era5\_to\_xarray(nc\_path, var\_name='tp'):  
 """Open NetCDF with xarray and decode, return xarray.DataArray (tp = total\_precipitation)  
 ERA5 'total\_precipitation' units are meters — convert to mm (multiply by 1000).  
 """  
 ds = xr.open\_dataset(nc\_path, chunks={'time': 24, 'latitude': 256, 'longitude': 256})  
 # variable name may differ depending on source; try to find it  
 if 'total\_precipitation' in ds:  
 da = ds['total\_precipitation']  
 elif 'tp' in ds:  
 da = ds['tp']  
 else:  
 # pick the first data variable  
 var = list(ds.data\_vars)[0]  
 da = ds[var]  
 da = da \* 1000.0 # m -> mm  
 # give rioxarray georef for saving  
 da = da.rio.write\_crs("EPSG:4326", inplace=True)  
 return da  
  
  
def save\_da\_as\_cog(da, out\_tif, resampling=Resampling.bilinear):  
 """Save an xarray DataArray (single time slice) as a COG using rio-cogeo profiles."""  
 ensure\_dir(os.path.dirname(out\_tif))  
 # first save a temporary GeoTIFF  
 tmp = out\_tif + '.tmp.tif'  
 da.rio.to\_raster(tmp)  
  
 profile = cog\_profiles.get('deflate')  
 config = {  
 'GDAL\_TIFF\_OVR\_BLOCKSIZE': '128'  
 }  
 cog\_translate(  
 tmp,  
 out\_tif,  
 profile,  
 InMemory=False  
 )  
 os.remove(tmp)  
 print('Wrote COG:', out\_tif)  
  
  
  
  
  
  
  
  
  
  
  
  
# --- SECTION 3: Small example workflow  
  
DATA\_DIR = os.path.abspath('data')  
ensure\_dir(DATA\_DIR)  
  
# Example bbox: small area around a city — change as needed  
# Format for CDS: [north, west, south, east]  
BBOX = [13.1, 77.4, 12.8, 77.8] # example: part of Bangalore, India  
START = datetime(2020, 8, 1)  
END = datetime(2020, 8, 2)  
  
era5\_nc = os.path.join(DATA\_DIR, 'era5\_tp\_example.nc')  
  
print('NOTE: If you do not want to download ERA5, skip to the OSM section and use sample NetCDF if you have one')  
  
# Uncomment to download (requires cdsapi and account). For many users, skip this step locally.  
download\_era5\_precip(START, END, BBOX, era5\_nc)  
  
# If you downloaded, open era5; else try to skip gracefully  
if os.path.exists(era5\_nc):  
 da = open\_era5\_to\_xarray(era5\_nc)  
 print('ERA5 data loaded with shape:', da.shape)  
 # For a quick demo, pick the first time slice  
 first = da.isel(time=0)  
 out\_cog = os.path.join(DATA\_DIR, 'era5\_tp\_t0\_cog.tif')  
 save\_da\_as\_cog(first, out\_cog)  
else:  
 print('ERA5 NetCDF not found; skipping raster generation. Place a NetCDF at', era5\_nc, 'or run CDS download')  
  
  
  
  
  
  
  
# --- SECTION 4: Fetch OSM data for the bbox using OSMnx  
print('Downloading OSM data for bbox', BBOX)  
  
# OSMnx expects (north, south, east, west)  
north, west, south, east = BBOX[0], BBOX[1], BBOX[2], BBOX[3]  
  
# Overpass query: fetch both buildings and highways  
custom\_filter = '["building"]["highway"]'  
  
features\_gdf = ox.features\_from\_bbox(  
 north, south, east, west,  
 custom\_filter=custom\_filter  
)  
  
# Separate buildings and roads into different GeoDataFrames  
buildings\_gdf = features\_gdf[features\_gdf['building'].notnull()].copy()  
roads\_gdf = features\_gdf[features\_gdf['highway'].notnull()].copy()  
  
print(f'OSM buildings: {len(buildings\_gdf)}, road features: {len(roads\_gdf)}')  
  
# Save to GeoJSON for quick serving  
osm\_dir = os.path.join(DATA\_DIR, 'osm')  
ensure\_dir(osm\_dir)  
  
buildings\_fn = os.path.join(osm\_dir, 'buildings.geojson')  
roads\_fn = os.path.join(osm\_dir, 'roads.geojson')  
  
buildings\_gdf.to\_file(buildings\_fn, driver='GeoJSON')  
roads\_gdf.to\_file(roads\_fn, driver='GeoJSON')  
  
print(f'Saved OSM assets to {osm\_dir}')  
  
  
  
  
  
  
  
  
  
  
  
  
  
# --- SECTION 5: Small FastAPI app to serve the COG and GeoJSON  
app = FastAPI()  
  
COG\_PATH = os.path.join(DATA\_DIR, 'era5\_tp\_t0\_cog.tif')  
  
  
@app.get('/osm/buildings')  
async def get\_buildings():  
 if not os.path.exists(buildings\_fn):  
 raise HTTPException(status\_code=404, detail='Buildings not found')  
 return FileResponse(buildings\_fn, media\_type='application/geo+json')  
  
  
@app.get('/osm/roads')  
async def get\_roads():  
 if not os.path.exists(roads\_fn):  
 raise HTTPException(status\_code=404, detail='Roads not found')  
 return FileResponse(roads\_fn, media\_type='application/geo+json')  
  
  
@app.get('/tiles/precip/cog')  
async def get\_cog():  
 if not os.path.exists(COG\_PATH):  
 raise HTTPException(status\_code=404, detail='COG not found')  
 return FileResponse(COG\_PATH, media\_type='image/tiff')  
  
# To run the server, run 'uvicorn digital\_twin\_mvp:app --reload --port 8000'  
  
  
  
  
  
  
  
  
  
  
  
# --- SECTION 6: A small Folium client to show the COG overlay + OSM assets  
# Note: folium cannot natively consume COGs; for a quick demo we convert a COG to a simple image overlay (low-res) or use Tile server.  
  
if os.path.exists(COG\_PATH):  
 # Read a small thumbnail from the COG  
 import numpy as np  
 with rasterio.open(COG\_PATH) as src:  
 arr = src.read(1, out\_shape=(int(src.height/8), int(src.width/8)))  
 bounds = src.bounds  
 arr = np.nan\_to\_num(arr)  
 # normalize for display  
 vmin, vmax = np.percentile(arr, (2, 98))  
 arr\_norm = (arr - vmin) / (vmax - vmin + 1e-6)  
 arr\_norm = np.clip(arr\_norm, 0, 1)  
 # convert to RGB by applying a simple colormap (matplotlib is optional)  
 try:  
 import matplotlib.pyplot as plt  
 cmap = plt.get\_cmap('viridis')  
 rgba = cmap(arr\_norm)  
 rgba\_img = (rgba[:, :, :3] \* 255).astype('uint8')  
 # save PNG  
 png\_path = os.path.join(DATA\_DIR, 'precip\_overlay.png')  
 import imageio  
 imageio.imwrite(png\_path, rgba\_img)  
 print('Wrote preview image to', png\_path)  
  
 # create folium map  
 m = folium.Map(location=[(bounds.top + bounds.bottom) / 2, (bounds.left + bounds.right) / 2], zoom\_start=12)  
 folium.raster\_layers.ImageOverlay(name='Precip (t0)', image=png\_path,  
 bounds=[[bounds.bottom, bounds.left], [bounds.top, bounds.right]], opacity=0.6).add\_to(m)  
 folium.GeoJson(buildings\_fn, name='Buildings').add\_to(m)  
 folium.LayerControl().add\_to(m)  
 folium\_out = os.path.join(DATA\_DIR, 'map\_preview.html')  
 m.save(folium\_out)  
 print('Saved Folium preview to', folium\_out)  
 except Exception as e:  
 print('Could not create image overlay preview:', e)  
else:  
 print('No COG available — skipping map preview generation')  
  
print('\nDone. To run the API server:')  
print(" uvicorn digital\_twin\_mvp:app --reload --port 8000")  
print("Then open data/map\_preview.html (if created) or visit http://localhost:8000/osm/buildings")