Chapter 2: Plant Demonstrator: A Case Study

Before You Start

What you are assumed to know. You write software for a living, you are comfortable with HTTP APIs, JSON, a time-series or relational database, and reading Python you did not write, and you have seen a system that talks to hardware. Two terms are used throughout: the **physical twin (PT)** is the real system being twinned, and the **digital twin (DT)** is the software kept in step with it, which answers questions about it and, in time, controls it.

What you are not assumed to know. Anything about electronics beyond plugging a cable into a socket, anything about serial buses, anything about soil or plants beyond having kept one alive or failed to, and anything about control. Where this chapter needs a term from one of those fields it defines it at first use and then keeps using it.

What this chapter covers. This book needs one concrete physical system that every later chapter can point at, small enough to sit on a desk and cheap enough that a reader could build it. This chapter introduces that system: a potted plant, watered by a small pump, under a Raspberry Pi that reads soil, air and light sensors and exposes everything over an HTTP interface. It is a real rig that others have built and documented, not an invented one; a pointer to its build documentation is in Further Reading, and this chapter stays at the level of the overview. It describes the rig as a block diagram, identifies which parts are sensors and which are actuators, walks up the four protocols that carry a soil reading from a probe to a JSON document, draws the line where the PT's software boundary sits, works two watering strategies and the pump calibration both depend on, and closes by asking what a DT of a plant would be for and what it would have to measure. Along the way it writes down a small mathematical model of each side: the pot, pump and controller below the boundary, and the twin that predicts and checks above it.

What this chapter deliberately does not cover. How to assemble the rig or which parts to order: this is an overview, and the build documentation is where those details live. Plant physiology or soil science beyond the minimum needed to motivate a metric. Electronics below the block-diagram level. And the DT as software – its architecture, how its model is calibrated, how that calibration is kept honest – which is the subject of the chapters that follow. This chapter ends where the DT's equations are on paper and its code is not.

Learning objectives. By the end of this chapter you will be able to:

1. **Describe** the plant controller as three groups of blocks, **classify** each block as a sensor, an actuator or neither, and **name** which of its two buses each one hangs on.
2. **Trace** one soil-moisture reading from the probe to a JSON document in a browser, naming the protocol in use at each hop and what each protocol adds.
3. **Explain** why a DT should depend on the controller's REST API and on nothing beneath it, and **relate** that boundary to the physical adapter of a general-purpose DT framework.
4. **Compute** a pump's calibration from a table of measurements, **convert** a dose in millilitres into a relay-on time, and **state** what goes wrong when calibration is skipped or stale.
5. **Compare** a fixed-schedule watering strategy with a target-moisture strategy by how long a plant stays too dry before anything acts, and **write** the REST call that switches the rig from one to the other.

6. **Propose** three or four metrics a plant DT should monitor and control, each with a unit and a reason.
7. **Write down** the equations of the physical twin – drying, dose response, pump time, delivery – and of the digital twin – dose rule, prediction, residual – and **run** them by hand across one check interval.

Every section that follows serves at least one of those seven, and the exercises at the end say which.

Physical Twin

The PT is *the real system being twinned*. For this book that system is a plant in a pot. Start with the parts you can see.

What is on the desk

On the desk there is a potted plant with a plastic tube running into its soil and a small probe pushed into the soil beside it. The tube comes from a submersible pump sitting in a water tank. Next to the pot is a Raspberry Pi 5 with its power supply, a USB dongle, a small relay board, a second power supply for the pumps, and a chain of thumbnail-sized sensor boards joined by short cables. That is the whole PT. This chapter says "the controller" for the Raspberry Pi and the software on it, and "the rig" or "the PT" for the whole thing including plant, pot, tank and tubing.

Two design goals shape everything below. The rig is built from **off-the-shelf parts**, so that it can be rebuilt on another desk without any electronics design. And it is meant as a **physical twin test kit for digital-twin work**: each plant can be watered on a schedule or on demand, and its parameters, along with those of its surroundings, are continuously measured, recorded, and exposed over an HTTP interface for a DT to consume. The first goal is why the parts list is short; the second is why the software boundary in the Physical Interface section exists at all.

It is not the only rig of its kind, and knowing the neighbours helps you see what is a design choice and what is a convention. GreenhouseDT, a community exemplar from the University of Oslo, twins a small greenhouse where plants, sensors and irrigation pumps together make up the physical rig, run by Raspberry Pi minicomputers, and its authors note that sensors and actuators have to be calibrated for the individual asset [1]. A study of reproducibility in DT research, built around a small robot car, argues that research twins should be rebuildable by others, and describes ocean-observation platforms whose sensors and actuators sit on a fieldbus controller over an RS-485 bus with a Raspberry Pi exchanging data and commands with it [2]. The rig in this chapter is a member of that family: cheap single-board computer, commodity sensors, one serial bus for the things that need distance, a database, an API.

The block diagram

Figure 1 is the rig as three groups and the plant they all end at. Read it top to bottom.

The **controller** group is the Raspberry Pi 5, its 27 W USB-C power supply, and a USB-to-RS-485 adapter. The Raspberry Pi runs the controller software and is the only computer in the rig. Its wireless network connection is how anything outside the desk reaches it.

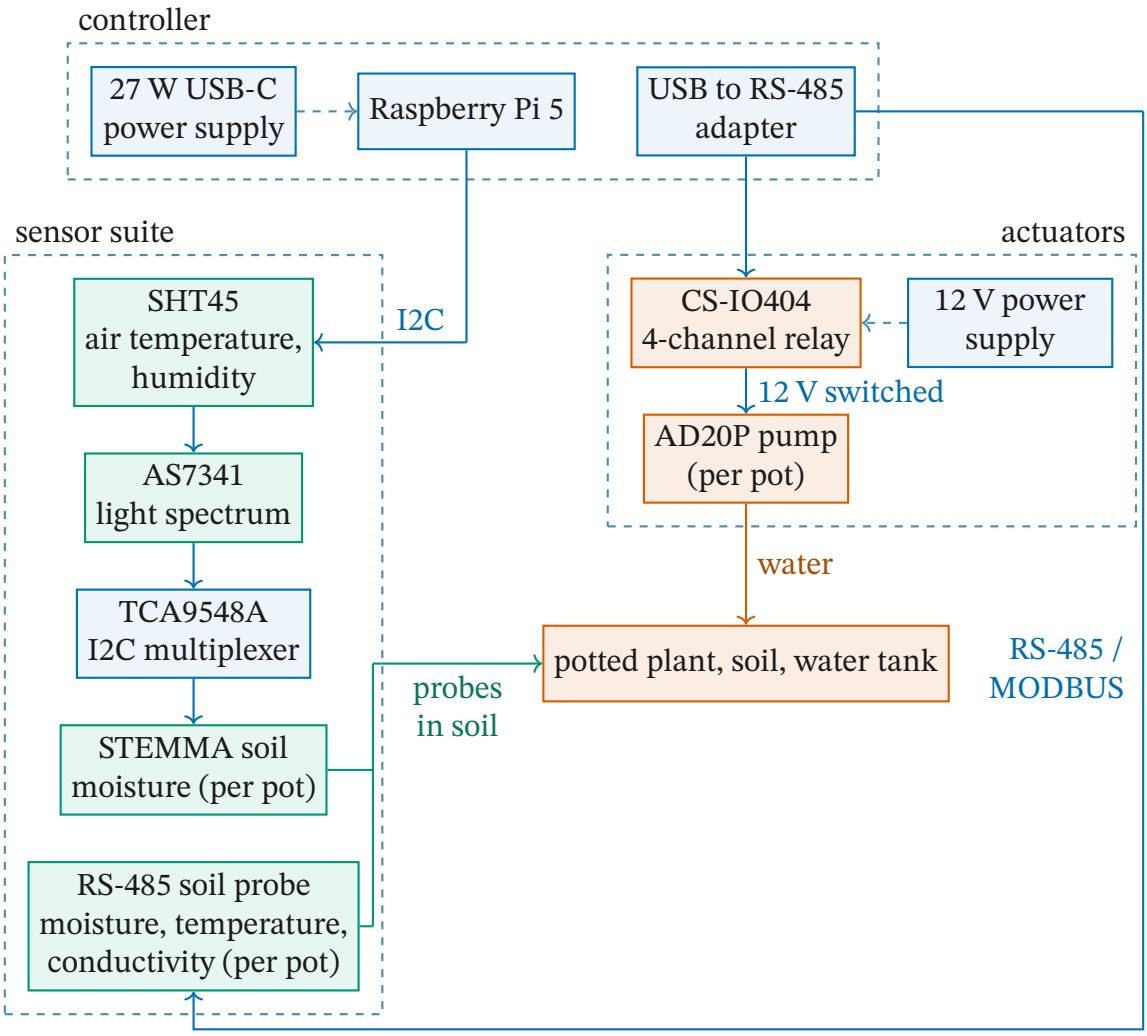


Figure 1: The plant controller as three groups – controller, sensor suite, actuators – and the potted plant every path ends at. Green blocks are sensors, orange blocks are actuators, blue blocks are neither. The I2C chain runs down the left, the RS-485 bus down the right.

The **sensor suite** is everything that turns a physical quantity into a number. Two sensors watch the surroundings, which the software calls the greenhouse even when the surroundings are an office: an SHT45 module measures air temperature and relative humidity, and an AS7341 module measures light as a spectrum across nine wavelength bands from violet to near infrared. Per pot there is one of two soil probes. The cheaper option is a capacitive STEMMA soil-moisture sensor, which reports moisture alone. The recommended option for DT work is an RS-485 soil probe that reports moisture, soil temperature and electrical conductivity from one set of tines. Because every STEMMA sensor answers to the same address, a TCA9548A multiplexer sits between the chain and the soil sensors so that up to seven of them can share one bus.

The **actuators** are everything the controller can command to change the world. There is exactly one kind: an AD20P submersible pump per pot, switched on and off by one channel of a CS-IO404 four-channel relay module. A 12 V power supply feeds the relay board and, through it, the pumps.

Everything else in the picture – the two power supplies, the RS-485 adapter, the multiplexer,

the cables, the tubing, the tank, the pot – is neither sensor nor actuator. That distinction is worth being strict about, because a later chapter's model of the rig has an input for every actuator and an output for every sensor and nothing else, and a part that is neither does not appear in the model at all. Table 1 makes the classification explicit.

Table 1: Every block in the rig, classified. Only the green and orange rows of the figure appear in a model of the rig; the rest is plumbing. The two power supplies are left out: they power blocks and are not themselves part of the twin's picture.

Block	Group	Sensor, actuator or neither	Bus	What it does
Raspberry Pi 5	controller	neither	–	runs the software and hosts the API
USB-to-RS-485 adapter	controller	neither	RS-485	lets the Raspberry Pi speak MODBUS
SHT45 module	sensor suite	sensor	I2C	measures air temperature and relative humidity
AS7341 module	sensor suite	sensor	I2C	measures the light spectrum in nine bands
TCA9548A multiplexer	sensor suite	neither	I2C	lets several same-address sensors share the bus
STEMMA soil sensor	sensor suite	sensor	I2C	measures soil moisture, one per pot
RS-485 soil probe	sensor suite	sensor	RS-485	measures soil moisture, temperature and conductivity, one per pot
CS-IO404 relay module	actuators	actuator	RS-485	switches 12 V to each pump
AD20P pump	actuators	actuator	–	moves water from tank to pot
tank, tubing, pot, plant	–	neither	–	the thing being twinned

Read the table down its third column: four kinds of sensor, two kinds of actuator, and four things that are neither. Now read the fourth: there are only two buses, and the RS-485 one carries both a sensor and an actuator. That is the fact to hold on to for the next section.

The two buses, and the power

The sensor boards along the left of Figure 1 are joined in a **daisy chain**: a four-wire cable runs from the Raspberry Pi's header to the SHT45, a second from the SHT45 to the AS7341, a third on to the multiplexer, and from the multiplexer one cable per soil sensor in a star. The four wires are power at 3.3 V, ground, a data line and a clock line. This is an I2C bus,

and the next section says what that means; for now, note that the whole chain is powered from the Raspberry Pi and that it is short – centimetres, not metres.

The devices along the right hang on a different bus. From the RS-485 adapter run two wires, twisted together, to the relay board's `A+` and `B−` terminals; the RS-485 soil probes are spliced onto the same two wires wherever convenient. Good practice asks for the twist, and for a 120 ohm resistor across each end of the pair, for a reason the next section gives. This bus is long – metres are fine – and it is the one on which the controller sends commands, not only reads values.

Power follows three separate paths, and keeping them separate in your head prevents the commonest wiring mistake. The Raspberry Pi and the whole I2C chain run from the 27 W supply at 3.3 V and 5 V. The RS-485 soil probes take 5 V from the RS-485 adapter's own power terminals. The relay board and the pumps take 12 V from the second supply, and each relay channel does nothing more than connect that 12 V to one pump for as long as the controller tells it to. Nothing on the I2C side ever touches 12 V.

What the sensors actually report

A block diagram tells you what is measured; a DT will want to know in what units and how often. Table 2 lists every quantity the rig produces, as the controller software labels it.

Table 2: The quantities the rig produces, named as the controller stores them. The `watering` row is not a sensor reading; it is the controller's own record of what it did.

Unit	Parameter, as stored	Reported as	Time between reads
greenhouse	`air_temperature`	degrees Celsius, with a stated confidence of plus or minus 0.2 degrees at 95 per cent	configurable
greenhouse	`air_humidity`	per cent relative humidity	configurable
greenhouse	`light_level_415nm` through `light_level_910nm`	photon flux, photons per second, nine bands	15 s
plant (STEMMA)	`moisture`	per cent of the sensor's full scale	configurable, 10 s in the shipped example
plant (RS-485 probe)	`soil_moisture`	per cent	configurable
plant (RS-485 probe)	`soil_temperature`	degrees Celsius	configurable
plant (RS-485 probe)	`soil_electrical_conductivity`	microsiemens per centimetre	configurable

Unit	Parameter, as stored	Reported as	Time between reads
plant	`watering`	one event per pump run, with its dose in millilitres	on each dose

Two rows deserve a second look. The STEMMA sensor's "per cent" is not a volumetric water content: the software reads a ten-bit number between 0 and 1023 from the probe and scales it linearly to a percentage, so a raw reading of 512 becomes $100 \times 512/1023 = 50$. The source comments say so and call for a calibrated transform to replace it. Treat that per cent as a relative scale – wetter is higher – and nothing more, until the chapter on calibration gives you a reason to trust it further. And the `watering` row is different in kind from every other: it is not something the rig observed but something the rig *did*. A DT that forgets the difference will build a model of the pot from a column that is really a model of the pump schedule.

The "time between reads" column is a design decision, not a property of the hardware. The chapter of *The Engineering of Digital Twins* that surveys sensing and communication for twins puts the trade-off plainly: a sampling frequency chosen for the worst case generates far more data than needed, while one chosen for the typical case can become inadequate when the environment changes, and the choice of sensors and their characteristics has to be made early, at design time [3]. For soil in a pot, which changes over hours, a read every ten seconds is generous. For a greenhouse whose door opens, fifteen seconds on the light sensor is about right.

A mathematical model of the physical twin

Everything above is a description. A DT needs the PT in a second form: a handful of equations that say how the numbers the rig reports change with time and with what the controller does. This subsection writes that model down, in the order a reader can run it by hand – state and parameters first, then the pot, then the controller and the pump it drives. It is deliberately the simplest model that closes the loop; the calibration chapter replaces its constants with fitted ones, and its straight lines with curves where the data demand it.

State, parameters and initial condition. The model tracks one quantity that changes, the soil moisture m, on the same per-cent scale the probe reports. Everything else is a parameter: a number that holds still over a day and that a calibration sets. Table 3 lists them.

Table 3: The physical twin's state and parameters. There are two pairs of pump constants because the controller acts on the pair it has stored while the pump obeys the pair it really has; the two agree right after a calibration and drift apart afterwards.

Symbol	Meaning	Unit	Where it lives
m	soil moisture, the state; needs a starting value $m(0)$	per cent	the pot, read by the probe

Symbol	Meaning	Unit	Where it lives
d	drying rate: what the pot loses in an hour with no dose	percentage points per hour	the pot
g	dose response: what one millilitre adds to the reading	percentage points per millilitre	the pot
a, b	the calibration slope and offset the controller has stored	seconds per millilitre; seconds	the controller's configuration
a_{true}, b_{true}	the slope and offset the pump really has today	seconds per millilitre; seconds	the pump, tank and tubing

The pot dries. With no dose, moisture falls at the drying rate. After h hours,

$$m(t + h) = m(t) - d\,h \tag{1}$$

Equation 1 is the whole of the pot's physics as this chapter needs it. A constant d is a simplification a reader can see through at once – a pot dries faster on a hot bright afternoon than at night, and the greenhouse unit measures exactly the quantities that would let d vary – but it is the right first model, because it is the one whose parameter the rig's own data can recover, as the Plant Digital Twin section shows.

A dose raises it. A volume of water V_{del} that actually arrives in the pot lifts the reading by g points per millilitre:

$$m_{\text{after}} = m_{\text{before}} + g\,V_{\text{del}} \tag{2}$$

The subscript on V_{del} matters. Equation 2 takes the millilitres that reach the soil, not the millilitres a schedule asked for; the next two equations connect the two, and the gap between them is where a stale pump hides.

The controller turns a dose into a time. The relay knows only *on* and *off*, so the controller converts a requested dose V into a pump time t with a straight line through the constants it has stored:

$$t = a\,V + b \tag{3}$$

The slope a is the reciprocal of the pump's flow rate; the offset b is the time the pump runs before the first drop arrives, spent lifting water up the tube. Where those two numbers come from, and how to read them, is the Watering Strategies section's worked example; here they are the controller's belief about its pump.

The pump delivers by its own constants. Held on for t seconds, the pump delivers whatever its real slope and offset say, which is Equation 3 rearranged with the true pair in place of the stored one:

$$V_{\text{del}} = (t - b_{\text{true}})/a_{\text{true}} \tag{4}$$

When the stored constants equal the true ones, Equation 3 and Equation 4 undo each other and the pot receives exactly the dose that was asked for. When they do not, every dose is short or long by an amount that nothing in the controller's own records shows.

Running it. Those four equations and a starting value $m(0)$ are the whole physical twin on paper. To run it for a day, step the clock an hour at a time: read m; deliver any dose due at that hour through Equation 3 and Equation 4 and apply Equation 2; then dry for an hour by Equation 1. The Watering Strategies section supplies the doses – a fixed schedule, or a rule that reads m – and the Plant Digital Twin section puts a model of the same shape on the other side of the API and asks what happens when the two disagree.

Low-level Protocols

Objective 2 asks you to follow one soil-moisture reading from the tines of a probe to a JSON document in a browser. Four protocols carry it, and each one adds something the one below could not. Take them in order.

I2C: many chips, two wires, one master

I2C (Inter-Integrated Circuit) is the bus for the sensors sitting next to the controller. It has two signal wires – **SDA** carries data, **SCL** carries a clock – plus power and ground, which is why every cable in the daisy chain has four conductors. The Raspberry Pi is the only **master**: it starts every exchange. Every other chip is a **peripheral** with a fixed **address**, and an exchange begins with the master calling out an address so that exactly one chip answers.

The address is the whole idea, and Figure 2 shows it for a rig with three pots. The air sensor and the light sensor each have an address of their own, so they can share the wire. The three soil sensors do not: every sensor of that kind leaves the factory with the same address, so on one wire all three would answer at once. The multiplexer solves that. It has an address of its own and eight numbered ports; the controller first tells the multiplexer which port to open, then addresses the soil sensor behind it. "Pot 2's moisture" is therefore not one address but a pair – the multiplexer's port, then the sensor's address – and that pair is what the controller's configuration records for each pot. The numbers in the figure are the chips' defaults; what matters is not their values but the rule they obey: every device the master wants to reach must have a name no other device on that wire shares.

I2C is built into nearly every small sensor chip, so the controller's Python code never touches the bus directly: a per-chip library turns "read the air sensor" into a method call. Its one weakness is distance – centimetres, not metres – which is what the second bus is for.

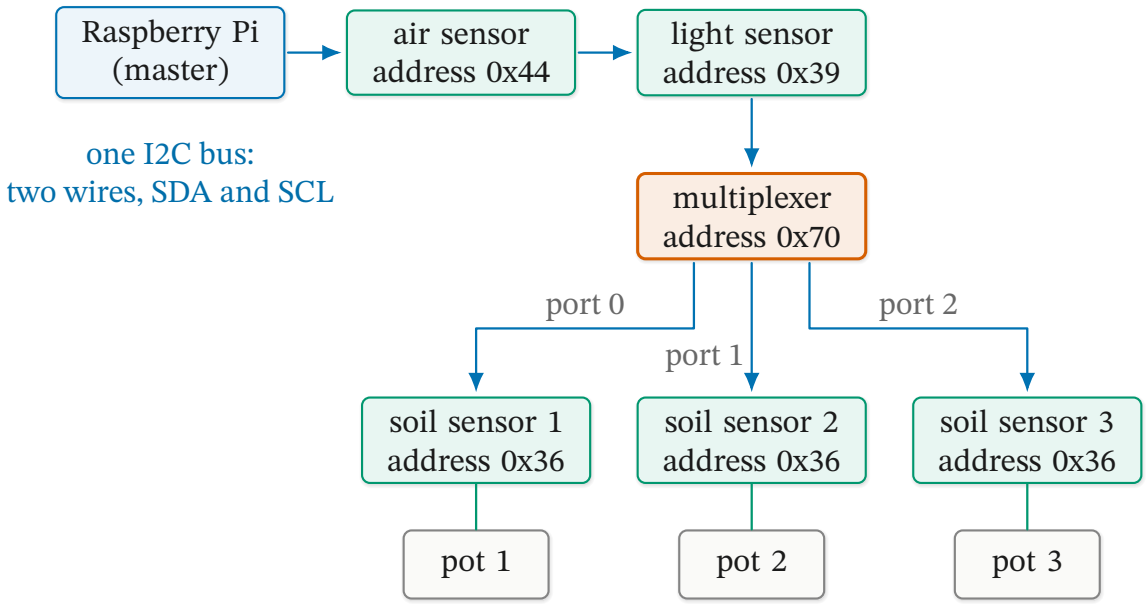


Figure 2: The I2C chain for three pots. Each device the controller talks to needs an address no other device on the wire shares; the three identical soil sensors get theirs by sitting behind different ports of the multiplexer.

MODBUS RTU over RS-485: distance, and commands

The second bus reaches the relay board and the soil probes in the pots, which may be metres away, and it is the one bus on which the controller gives orders as well as asking questions. Two standards are stacked here, and it pays to keep them apart.

RS-485 is the wire. It is an electrical standard for a **differential** pair: two twisted wires carry the same signal with opposite polarity, so noise picked up along the cable affects both equally and cancels at the receiver. That is what buys tens of metres where I2C manages centimetres; it is why the pair is twisted, and why a 120 ohm resistor across each end of a long bus is good practice – it stops the signal reflecting back from the ends. Many devices share the one pair, and at any moment one talks and the rest listen.

MODBUS is the language spoken on the wire. It is a request-and-reply protocol from industrial automation, and it comes in three variants – TCP/IP, RTU and ASCII – of which RTU is the one for serial lines, with one master and many slaves [4]. It is old, small and everywhere: the ISO/IEC reference architecture for the Internet of Things (IoT) lists MODBUS beside Ethernet and Wi-Fi among the data-transferring capabilities an IoT system may be built on [5]. The controller is the client. Each frame it sends names a **device id**, a function, and an address inside that device; the one device with that id acts and answers, and nobody else speaks.

MODBUS names the data inside a device in two ways, and the rig uses both:

- A **coil** is one on/off output. The relay module's four channels are coils 0 to 3, and "run pump 2" is a write of the value `true` to coil 1 at the relay module's device id.
- A **holding register** is one sixteen-bit number. Each soil probe exposes moisture, temperature and conductivity in three consecutive registers, which the controller reads and scales.

Figure 3 shows the same three-pot rig from this bus's point of view. Every device on the wire has a device id no other device shares; inside a device, coils and registers are numbered. Two probes with the same id would both answer, exactly as two identical soil sensors did on I2C – and here there is no multiplexer to rescue them, so a new probe's id is changed once, at setup, before it joins the bus. Again the numbers are illustrative; the rule is that an address is unique on its wire.

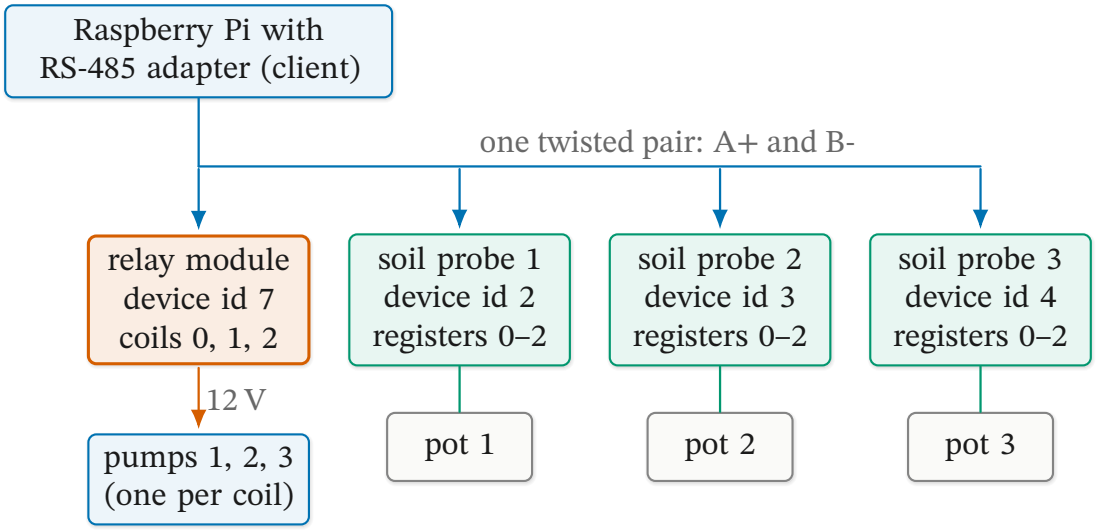


Figure 3: The RS-485 bus for three pots. One cable and four devices, each with a device id of its own; inside the relay module the three pumps are coils 0 to 2, and inside each probe the three quantities are registers 0 to 2.

The distinction to carry forward: on I2C the controller only *reads*. On MODBUS it reads registers *and writes coils*, and writing a coil is the single point at which software in this rig changes the physical world.

InfluxDB: readings become a time series

Every reading and every watering event ends up in a database on the Raspberry Pi, and the database is **InfluxDB 3**, a time-series database. A time-series database is one whose rows are timestamped values and whose queries are about time: the last hour, since Tuesday, the most recent hundred. The foundational chapter of *The Engineering of Digital Twins* describes InfluxDB as an open-source time-series database designed to handle large volumes of timestamped data, which is what makes it a common choice for monitoring and IoT work [6], and the same book's tutorial on an incubator twin uses one in exactly the rig's role: a Python service listens to the traffic between the controller and its low-level driver, then writes those messages into InfluxDB [7]. GreenhouseDT does the same, storing its sensor data in a time-series database the twin can query [1].

The storage layout is simple and worth knowing, because a DT will read it through the API. Every datum becomes a **point** with four parts: a **measurement** name, which is InfluxDB's word for a table; **tags**, which are indexed strings; **fields**, which are the values; and a timestamp. The controller keeps one measurement per unit and parameter, named `<unit>_<parameter>`, so the moisture reading for the plant named `plant_1` lands in `plant_1_moisture`, with the value, its units and, where the sensor states one, a confidence as its fields. A watering event goes to `plant_1_watering` with the dose in millilitres as its value.

Reading it back is a query in InfluxDB 3's SQL dialect, and the controller issues one shape of query only:

```
SELECT * FROM plant_1_moisture
WHERE time > '2026-09-01T00:00:00'
ORDER BY time DESC
LIMIT 100
```

The database keeps a **retention period** of seven days: older points are dropped, so the controller's memory card does not fill. That single setting has a consequence for a DT that the rest of this book will keep returning to – anything that wants more than a week of history has to copy it out before it goes.

HTTP: the reading leaves the desk

The last hop is the one you already know. **HTTP** (Hypertext Transfer Protocol) is the request-and-reply protocol of the web: a client opens a connection, sends a request naming a **method** (`GET` to read, `PUT` to replace) and a **resource** by its path, and the server sends back a **status code** and a body. A comparison of messaging protocols for IoT systems describes HTTP as supporting the request/response RESTful web architecture, addressing its resources by Uniform Resource Identifier in the same way the constrained-device protocol CoAP does [8], and that is the role it plays here: the controller runs an HTTP server on port 8099, and every reading the rig has ever taken is one `GET` away for anyone on the same network. The next section is about what that server exposes and why it is the most important boundary in the rig.

The four, side by side

Table 4 puts the four protocols beside each other on the axes that matter for a DT.

Table 4: The four protocols a soil reading passes through, from the probe's tines to a browser.

Protocol	Layer it lives at	Who talks to whom	Names things by	Reads, writes or both	Distance
I2C	wires on the desk	one master, addressed peripherals	7-bit address, plus multiplexer port	reads (in this rig)	centimetres
MODBUS RTU over RS-485	wires across the room	one client, addressed devices	device id, then coil or register number	reads registers, writes coils	tens of metres
InfluxDB 3	storage on the controller	the controller software, and anything with the token	measurement, tags, time	writes points, reads by SQL	the Raspberry Pi

Protocol	Layer it lives at	Who talks to whom	Names things by	Reads, writes or both	Distance
HTTP REST	the network	any client, one server	resource path, query parameters	`GET` reads, `PUT` replaces a schedule	anywhere on the network

Read the table down the fifth column. The only place in the entire stack where something is written *towards* the plant is the coil write on MODBUS, and the only place where an outsider can cause that write is the one `PUT` in the last row. Everything else is readings flowing away from the plant. That asymmetry is the whole design, and the next section is built on it.

Physical Interface

Objective 3 is a claim about software architecture, and it is worth stating before the mechanism: **a DT should talk to the PT through one software boundary, and to nothing below it.** In this rig that boundary is the HTTP REST API on port 8099. This section says what the API offers, why the boundary sits there, and what the general form of the idea looks like in a DT framework built for it.

Two actors

Two kinds of user meet this rig. The **implementer** has physical access to the controller and works on it through its operating system, usually over SSH: editing configuration files, running the setup utility, starting and stopping the software. The **remote connecter** – a person, or a computer running a DT – reaches the controller only over the network, through the REST API. The two never use the same door, and that separation is deliberate. The implementer can rewire a sensor, recalibrate a pump or change a database token; the remote connecter can read what has been measured and change when the plant is watered, and nothing else.

What the API offers

The controller's web layer is a small FastAPI application, and it publishes its own interactive documentation at `/docs`. The endpoints that matter are in Table 5.

Table 5: The REST API on port 8099. Seven endpoints read; exactly one writes.

Method and path	Returns or does	Notes
`GET /`	controller version, current time, link to `/docs`	a liveness check
`GET /sensing`	the list of units that have sensors	`greenhouse` plus each plant

Method and path	Returns or does	Notes
`GET /sensing/{unit}`	that unit's parameters, each with units, confidence and time between reads	the machine-readable form of the quantities table
`GET /sensing/{unit}/{parameter}`	the stored measurements, newest first	`?limit=` caps the count; `?since_timestamp=` filters by time
`GET /actuation`	the list of units that have a pump	
`GET /actuation/{unit}/watering_events`	every dose delivered, newest first	same `limit` and `since_timestamp`
`GET /actuation/{unit}/show_schedule`	the current watering schedule as JSON	
`PUT /actuation/{unit}/update_schedule`	replaces the schedule; returns 204 with no body	422 if the schedule is malformed

Three details of the table repay attention. The unit and parameter names in the paths are the same strings the database uses for its measurement names, so the API is a thin, named view over the time series, not a second data model. Unknown units and parameters return `404` with the list of valid ones in the body, and a malformed timestamp returns `400` with the expected format spelled out (`YYYYMMDD-hhmmss`, with everything after the date optional), so a client can be wrong and find out why without reading the source. And the single `PUT` does not run a pump; it replaces a *schedule*, and the controller's own scheduler then runs the pump. A remote connecter cannot say "pump now"; it can only say "from now on, water at these times with these doses". The Watering Strategies section shows what can be built inside that constraint.

Here is the reading Objective 2 asked you to trace, at the end of its journey. The request

```
GET http://plant-controller.local:8099/sensing/plant_1/moisture?limit=2
```

returns a JSON array of the two most recent points, each carrying the tags and fields the Low-level Protocols section described:

```
[
  {"time": "2026-09-07T10:31:40", "physical_unit": "plant_1",
   "parameter": "moisture", "value": 41.2, "units": "%",
   "confidence": "+-1.0000e+00 at 95.00%"},
  {"time": "2026-09-07T10:31:30", "physical_unit": "plant_1",
   "parameter": "moisture", "value": 41.3, "units": "%",
   "confidence": "+-1.0000e+00 at 95.00%"}
]
```

Ten seconds apart, because that plant's sensor is configured with $T_r = 10$. A capacitance in a probe became a ten-bit number over I2C, a percentage in Python, a point in InfluxDB, and two objects in a JSON array. Nothing in that array says "I2C" or "multiplexer port 5", and that is the point.

Why the boundary sits there

Figure 4 draws the rig as a stack of layers, with the API as the line a DT is allowed to touch.

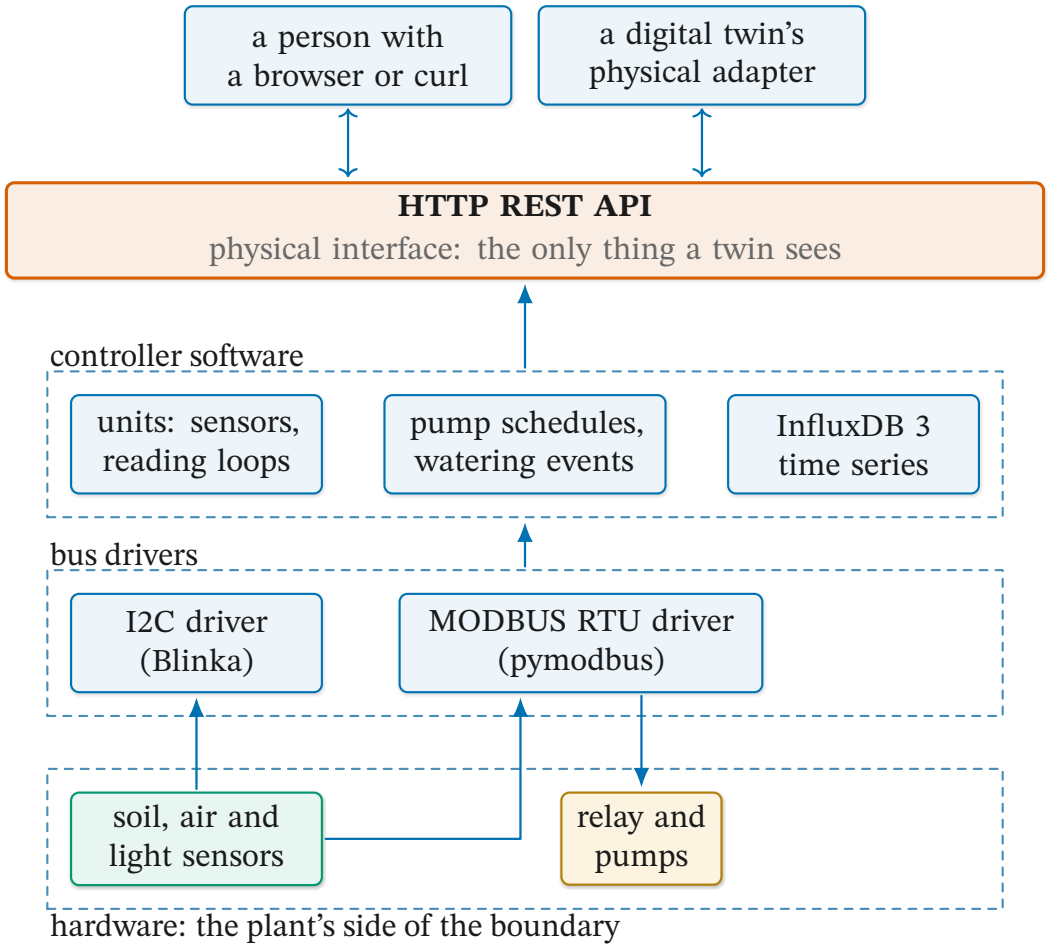


Figure 4: The rig as a stack. Each layer talks only to its neighbours, and a digital twin depends on exactly one of them: the REST API. Everything beneath it – controller software, bus drivers, hardware – can change without the twin noticing.

Three reasons put the line where it is, and all three are reasons you already know from ordinary software engineering, applied to a PT.

The layers beneath will change. This rig is itself the third revision of its design, and each revision swapped relay boards and sensors beneath an interface of the same shape. A DT written against `/sensing/plant_1/moisture` survives such a change; one written against a multiplexer port does not. The research literature on DT lifecycles makes the general point: a PT can change its telemetry frequency and its value ranges across the phases of its own lifecycle, and if those variations are not accounted for they complicate keeping the DT synchronised with it [9]. An API that publishes each parameter's units and time between reads, as this one does, is how a PT accounts for them.

The boundary is where responsibility changes hands. Below the line, the controller software owns correctness: it holds the bus lock, it converts registers to degrees, it decides that a dose of 100 ml means 3.5 s of relay time. Above the line, the DT owns *interpretation*: what a moisture of 41 per cent means for this plant, whether the last dose had the expected effect. The comparison of what a model expected with what the probe reported – the

residual, the difference between the two – is computed above the line from numbers produced below it, and neither side needs to know how the other does its job.

Safety lives below the line. The API cannot switch a relay. It can only replace a schedule, which the controller validates before accepting, and the controller alone decides how long a pump runs. A DT with a bug can water the plant at the wrong time; it cannot leave a pump running until the tank is empty and the floor is wet. Whether that limit is the right one is a design question the Watering Strategies section takes up, but that there *is* a limit, enforced on the PT's side of the line, is what makes an experimental DT safe to point at a real pot.

The general form: a physical adapter

Every rig draws this line somewhere, and the DT frameworks built for IoT give it a name. The **White Label Digital Twin (WLDT)** library, a Java framework for IoT twins, was designed to maximise modularity, re-usability and flexibility in mirroring physical objects into digital counterparts, and its authors describe extending a physical device's behaviour without altering it – a legacy-protocol example mirrors commercial light bulbs into digital replicas that answer external requests either by forwarding them to the devices or from a local cache, to cut response time and load on the physical objects [10]. A survey of IoT twin platforms summarises WLDT as a framework that runs at the edge or in the cloud and supports a wide range of communication protocols and normalised data formats [11].

The architectural idea behind it, which the same group later generalised as an *edge digital twin*, is that the twin is an active component that encapsulates the responsibility of talking to its physical counterpart, in a one-to-one relationship, and keeps the two in step – a process the authors call **shadowing**, supported by a module in charge of the physical layer's protocol heterogeneity [12]. In WLDT's vocabulary that module is the **physical adapter**: the one part of the twin that speaks the PT's protocols. A **digital adapter** on the other side exposes the twin to its consumers, and a **shadowing function** between them decides what the twin's state is and what a request means. An industrial case study built on the library shows the three in motion: configuration data reaches a machine's twin through its HTTP digital adapter, the shadowing function validates the request, and the physical adapter forwards it to the physical controller [13].

Set that beside Figure 4 and the correspondence is exact. The rig's REST API is the *physical interface*: the PT's side of the boundary. A DT's physical adapter for this rig is a small piece of code that knows seven `GET`s and one `PUT` – the twin's side of the same boundary. A later chapter gives the architecture of a DT; when it does, the physical adapter is where this chapter's material plugs in.

Watering Strategies

The rig's one actuator is a pump, and the only question a watering strategy answers is *when does it run, and for how long*. This section takes the two strategies the outline names, shows how each is expressed through the API, and then does the thing both depend on and neither can skip: calibrating the pump.

Strategy 1: a fixed daily schedule

The controller ships with one schedule type, called `daily`. A schedule is a JSON document with a `type` and a list of events, each with a time of day and a dose in millilitres:

```
{
  "type": "daily",
  "schedule": [
    {"time": "09:16:00", "dose": 100},
    {"time": "17:05:30", "dose": 120}
  ]
}
```

The scheduler sorts the events, sleeps until the next one, runs the pump for that dose, records a watering event, and repeats; after the last event of the day it sleeps until the first event of tomorrow. That is the entire algorithm, and it is the `daily` module's whole job.

Installing that schedule for a plant is one request:

```
PUT http://plant-controller.local:8099/actuation/plant_1/update_schedule
Content-Type: application/json

{"type": "daily", "schedule": [{"time": "09:16:00", "dose": 100},
                               {"time": "17:05:30", "dose": 120}]}
```

which returns `204 No Content` on success and 422 with a reason if, say, a dose is written as a string. The controller writes the new schedule to disk, cancels the running scheduler, and restarts it with the new events, so the change takes effect immediately and survives a reboot.

This is the commonest shape of automation there is: the pump runs on a timer, the sensor readings are stored, and the decision "how much water, and when" never reads the reading. Panel (a) of Figure 5 draws the loop that does not close. There is nothing wrong with it as a starting point – houseplants have been kept alive this way for as long as there have been timers – but it makes the decision once, when someone writes the schedule, and then stops deciding.

Strategy 2: a target moisture, checked at fixed intervals

The second strategy closes the loop, and it can be built entirely on the API's side of the line. Every T_c hours, a small program – the beginnings of a DT, or a shell script; the API does not care – fetches the latest moisture m for a plant, compares it with a **target moisture** m_t, and decides a dose:

$$V = \min\bigl(V_{\max},\; k\,(m_t - m)\bigr) \tag{5}$$

when $m < m_t$, and no dose when $m \geq m_t$. Here k is a **gain**, in millilitres per percentage point of moisture deficit, and $V_{\max}$ is a cap the program will never exceed in one dose. Having chosen V, the program `PUT`s a one-event schedule whose time is a minute from now, waits for the watering event to appear at `/actuation/plant_1/watering_events`, and then `PUT`s an empty schedule so nothing else happens until the next check. Panel (b) of Figure 5 is that loop.

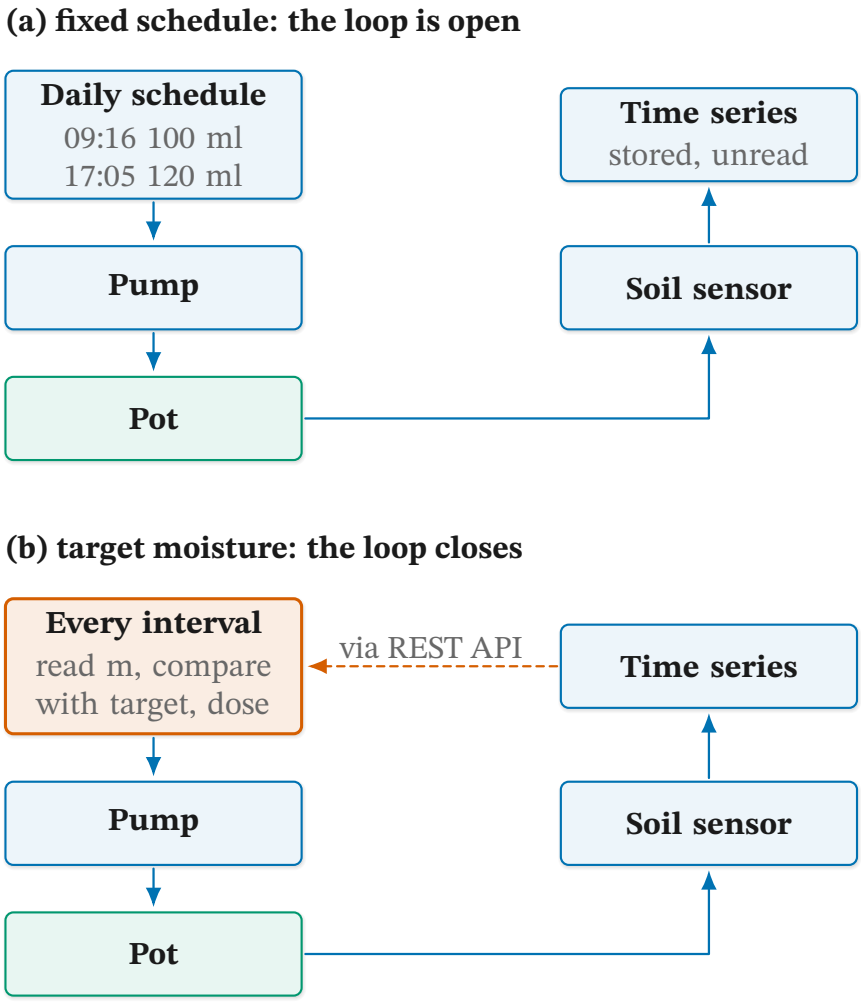


Figure 5: Two watering strategies on the same hardware. In (a) the schedule never reads the sensor, so the loop is open. In (b) a rule outside the controller reads the moisture through the REST API and rewrites the schedule, so the loop closes – through the API, never around it.

Two things about Equation 5 are choices, not consequences. The gain k decides how hard the rule pushes: too small and the plant stays under target for days, too large and one dry reading produces a flood. The cap $V_{\max}$ is the safety argument from the Physical Interface section moved up a level: whatever k says, one check delivers at most $V_{\max}$. Choosing k well needs the dose response g of the Physical Twin section's model – how much one millilitre raises the reading in *this* pot – and fitting g is the calibration chapter's subject; for now, a conservative k and a hard cap are what make the strategy safe to run unattended.

The literature on greenhouse twins does exactly this, one level up. GreenhouseDT's authors describe behavioural adaptation as the twin calibrating the frequency or duration of running its watering pumps to how quickly the soil in the pots gets dry [1], and their work on declarative lifecycle management gives the target a reason to move: a basil plant's target moisture should shift with its growth stage, with the paper's example holding a sick plant to a lower target than a healthy one, and therefore a different controller [14]. In this chapter's terms, m_t is a parameter of the plant's state, not a constant, and the DT is the thing that knows which state the plant is in.

Compare the two strategies by how long a dry pot goes unattended, and split that time into two gaps: the **detection gap** G_{d}, from the pot becoming too dry to something noticing, and the **response gap** G_{r}, from noticing to acting. Under the fixed schedule G_{d} is however long it takes a person to notice a drooping leaf – days. Under the target-moisture rule it is at most T_{c}, and G_{r} is the minute the schedule takes to fire. Setting $T_{\mathrm{c}} = 6$ puts the whole delay under seven hours for a fault the fixed schedule would take days to expose. The software has gone from watching the plant to acting on it, on a desk, for the price of a pump. The warning that comes with acting automatically applies too: a wrong m_{t}, or a probe that has slipped out of the soil and reads dry air, will now be acted on with nobody looking, which is why $V_{\max}$ exists and why a model of what the reading *should* be matters – the Plant Digital Twin section writes one down.

What both strategies stand on: pump calibration

Every dose above is a number in millilitres. The relay knows nothing of millilitres; it knows *on* and *off*. Somewhere the controller has to turn "100 ml" into "hold the relay closed for this many seconds", and that conversion is **pump calibration**. Get it wrong and every strategy in this section is wrong by the same factor, silently.

The controller's model is the straight line the Physical Twin section's model already wrote down, Equation 3: for a dose V in millilitres the pump time t in seconds is $t = a\,V + b$, with two calibration constants per pump, a **slope** a, in seconds per millilitre, and an **offset** b, in seconds. The shipped example configuration carries $a = 0.02$ and $b = 1.5$. Read Equation 3 physically before reading it mathematically. The slope is the reciprocal of the pump's flow rate: at $a = 0.02$ the pump moves $1/a$, about fifty millilitres, a second. The offset is the time the pump runs before any water reaches the pot – the time to lift water up the tube and push out whatever air is in it – and it is why a dose of $V = 0$ still costs $t = b$ seconds of pumping, and why very small doses are unreliable.

Both constants depend on things the pump's datasheet cannot know: the height the water is lifted, the length and bore of the tube, how far the pump sits below the surface, whether the tube drained back after the last run. That is why GreenhouseDT's authors say sensors and actuators need calibrating for the individual asset [1], and why the controller ships a calibration procedure rather than a table of pump models.

Example 1, worked in full: calibrating one pump The controller's setup utility runs the procedure interactively. Here is one run, with the reasoning at each step.

Step 1 – reproduce the installed geometry. The utility first asks that the pump be submerged in the full tank, connected to its relay channel, and that the outlet tube be positioned so it pumps into a measuring jug *at the same height relative to the tank as the pot*. The reasoning: b is mostly the cost of lifting water, so a calibration done with the tube lying in the sink is a calibration of a different pump.

Step 2 – measure the lift. The utility asks for the height from the water surface to the outlet, in centimetres. It uses the number to decide how long to wait between runs so the tube can drain back; for the example, the outlet sits 30 cm above the surface and the utility waits 3 s after each run.

Step 3 – run the pump for a ladder of durations and measure each volume. The utility runs the pump for 1.0 s, then 1.5 s, and so on in half-second steps up to 8.0 s, and after each run

asks how many millilitres landed in the jug, read to the nearest 10 ml. Fifteen runs; the reasoning for a ladder rather than one long run is that a single point cannot separate a from b. Table 6 records the example.

Table 6: One calibration run on the example pump. The first two runs delivered nothing and are discarded by the procedure, as its source specifies.

Run	Pump time (s)	Volume measured (ml)	Kept?
1	1.0	0	no – nothing reached the jug
2	1.5	0	no – nothing reached the jug
3	2.0	20	yes
4	2.5	50	yes
5	3.0	70	yes
6	3.5	100	yes
7	4.0	120	yes
8	4.5	150	yes
9	5.0	180	yes
10	5.5	200	yes
11	6.0	230	yes
12	6.5	250	yes
13	7.0	280	yes
14	7.5	300	yes
15	8.0	320	yes

Read the table from the top. The first two runs delivered nothing: the pump ran, but for less time than it takes to fill the tube, and the utility discards a zero reading rather than fit to it. From run 3 on, every extra half second delivers a further 20 to 30 ml. That constant increment *is* the slope, and the point where the line would cross zero volume *is* the offset.

Step 4 – fit the line. The utility fits pump time against measured volume by least squares – the source calls `numpy.polyfit` with degree one, with volume as the independent variable, since that is the direction Equation 3 is used in. On the thirteen kept rows the fit gives $a = 0.0197$ and $b = 1.56$. The reasoning for fitting time on volume, rather than volume on time and inverting: the constants are used to compute a time from a dose, so the errors should be minimised in the time direction.

Step 5 – read the result physically. A slope of $a = 0.0197$ is a flow of $1/a = 51$ millilitres a second, which is about three litres a minute – far more than a pot wants, and the reason doses are seconds long. An offset of $b = 1.56$ is a second and a half of pumping before the first drop arrives, which at that flow rate is $b/a = 1.56/0.0197 = 79$ millilitres of tube and lift that every run pays for and no run delivers. Both numbers are plausible for a 30 cm lift through a metre of 8 mm tube; a slope ten times larger or an offset that came out negative would mean a kinked tube or a mis-entered reading, and the utility prints the old and new constants side by side so exactly that comparison can be made.

Step 6 – save, with a date. The utility writes the new `slope` and `offset` into the plant's configuration file together with a `calibration_date`. The date is the important part. A pump calibration goes stale: the tank level drops and the lift grows, the tube stiffens, algae narrows the bore. Keeping a model valid is the cost most twin projects forget to budget for;

here it is in miniature, and the date is what lets a DT notice that the number it is trusting is three months old.

Step 7 – use it. A schedule asks for a dose of 100 ml. By Equation 3, the controller closes the relay for $t = 0.0197 \times 100 + 1.56 = 3.53$ seconds. The shipped constants would have given 3.50 s – close, but from a different pump on a different desk, and the 0.03 s difference is 1.5 ml on every dose, in the same direction, twice a day, for as long as nobody recalibrates.

Example 2, one step left to you: a dose from the target-moisture rule A plant's latest reading is $m = 32$ against a target $m_t = 40$, with gain $k = 10$ millilitres per point and cap $V_{max} = 150$. By Equation 5, the deficit is eight points, so $k\,(m_t - m) = 10 \times (40 - 32) = 80$, below the cap, and the dose is $V = 80$. **Your step:** using the constants fitted in Example 1, compute the relay time from Equation 3, then say what the dose and the relay time would have been had the reading come back as $m = 21$ instead – and which of the two numbers stopped the rule from doing what the deficit asked. A check appears in the exercise solutions.

Example 3, posed as a problem: a stale calibration Six weeks after Example 1 the tank has been topped up less often and sits lower, and the tube has been re-routed over the edge of a higher shelf. Nobody has recalibrated. Working out what happens to every dose under both strategies, which of the two strategies notices first and how, and what a DT would need to have recorded in order to raise the alarm, is the hardest exercise at the end of this chapter, and it is deliberately not worked here. The Physical Twin section's model has every equation it needs: the controller keeps using its stored a and b in Equation 3 while the pump obeys a changed b_{true} in Equation 4.

Plant Digital Twin

The chapter has so far described a PT and the software boundary a DT would use. It ends by asking the question any twin has to answer before it is worth building: what decision would a twin of this plant improve, and what would it have to measure to improve it? The full answer is the rest of the book. This section settles the motivation and the metrics.

Why moisture, and why minerals

A plant in a pot depends on two things the rig can influence, and the motivation for the DT is that both are easy to get wrong in both directions.

Water. Too little, and the plant wilts, stops growing, and eventually dies; too much, and the roots sit in water that has driven the air out of the soil, which kills them just as surely and more slowly. A pot is a small volume with no reserve, so the margin between the two is narrow and moves: a hot bright day dries the pot in hours that a cool dull one takes days to dry. A fixed schedule is right on average and wrong on most individual days.

Minerals. Plants take their nutrients from minerals dissolved in the soil water. In a pot those minerals come only from what was in the soil to begin with and what is added, and they leave by two routes: the plant takes them up, and every watering washes some out of the bottom of the pot. Too little and the plant starves; too much – typically from over-feeding – and the salt concentration in the soil water rises until the roots can no longer draw water against it. The rig's RS-485 probe measures the soil's **electrical conductivity (EC)**, which

rises with the concentration of dissolved salts, so EC is the rig's proxy for how much mineral is in the soil water. It is a proxy, not a measurement of any one nutrient, and the chapter keeps calling it that.

Both problems are recognised targets for agricultural twins. A survey of twin platforms for the IoT lists among the uses of smart-farming twins the evaluation of soil quality by monitoring moisture, temperature, organic matter and pollutants, and the forecasting of plants' irrigation requirements to reduce fertiliser and pesticide dosages and protect groundwater [11]. A field-scale twin of an irrigation pivot on an experimental farm collects soil moisture and the levels of different elements among its inputs [15], and a catalogue of DT architecture patterns takes its worked cases from agriculture and food [16]. The pot on the desk is those systems with the hectares removed.

The two paragraphs on water and minerals state horticultural common ground rather than results from the literature. Where a later chapter needs the actual water-retention curve of a potting mix, it will have to say where it got it.

Four metrics

A twin needs metrics because "the plant is doing well" cannot be checked, trended or acted on, and a quantity with a unit, a denominator and a way to measure it can. Here are four the rig can compute today from what it already stores, in Table 7, followed by the reason for each.

Table 7: Four metrics a plant DT can compute from the rig's stored data. The first and last are things to hold in a band; the middle two are what a model of the pot has to explain.

Metric	Symbol	Computed from	Unit	Monitor or control?
Time in the moisture band	f_{band}	the moisture series, a lower bound m_{lo} and an upper bound m_{hi}	fraction of readings	control target
Water delivered per day	W_{day}	the watering events	millilitres per day	monitor; the cost side
Drying rate between doses	d	two moisture readings after a dose	percentage points per hour	monitor; the model's input
Conductivity drift	$EC - EC_0$	the conductivity series against its value at potting	microsiemens per centimetre	control target, slow

Time in the moisture band is the quantity whose movement anyone who cares about the plant would recognise as the point. Define a band m_{lo} to m_{hi} for the plant's current stage, count the readings that fall inside it, and divide by all readings:

$$f_{\text{band}} = n_{\text{in}}/n \tag{6}$$

where n_{in} is the number of readings inside the band and n the number of readings. The fixed schedule of the previous section will score poorly on Equation 6 on any day that is not the average day; the target-moisture rule scores better by construction; and a DT that adjusts m_{t} to the plant's stage, as the declarative-lifecycle work proposes with its lower target for a sick basil than for a healthy one [14], is trying to push it towards one. It is a fraction, so it has a denominator built in, and it is computable from a single `GET`.

Water delivered per day is the cost side, and it is the metric the rig records most reliably because it is a record of the rig's own actions. Sum the doses in the watering events for one day, $W_{\text{day}} = V_1 + V_2 + \cdots + V_j$; the fixed schedule above gives $W_{\text{day}} = 220$ every day by definition, and the target-moisture rule gives a number that varies with the weather, which is the point. On a desk the cost of water is nothing, and the metric still earns its place, because it is the input to the next one.

Drying rate between doses is the first thing a *model* of the pot has to get right. Take the moisture just after a dose has soaked in and the moisture just before the next one, and divide by the time between them:

$$d = \frac{m(t_1) - m(t_2)}{t_2 - t_1} \tag{7}$$

If the reading is $m = 44$ an hour after the morning dose and $m = 36$ eight hours later, then $d = (44 - 36)/8 = 1.0$ percentage point an hour. Equation 7 is not a control target – you cannot set the drying rate – but it is what the model has to predict from air temperature, humidity and light, all of which the greenhouse unit already measures, and it is what the residual is computed from – the next subsection writes it down: the model says the pot should have dried by so much, the probe says it dried by this much, and the difference is the residual r. GreenhouseDT's managing subsystem relies on model-based forecasts to choose how it drives its greenhouse's actuators [1]; the drying rate is the smallest such prediction this rig supports.

Conductivity drift is the slow metric. Record the RS-485 probe's conductivity when the plant is potted, EC_0, and watch the difference $EC - EC_0$ over weeks. Falling drift says minerals are being used or washed out faster than they are replaced; rising drift says salts are accumulating. It is a control target in the sense that a person – not, in this rig, the controller – acts on it by feeding or by flushing the pot, so its delay is dominated by G_{r}, and the warning from the Watering Strategies section applies: a DT that shortens G_{d} for a decision nobody acts on has done nothing. That is a reason to keep it, not to drop it: it is the metric that will teach you, in a later chapter, the difference between monitoring and control.

A mathematical model of the digital twin

The metrics say what the twin watches. Its model says what it *expects*, and the difference between the two is the twin's whole advantage over a schedule. This subsection writes the DT's side down in the same order as the PT's: parameters first, then the equations, then how the two models run together across the API.

What the twin knows. The twin sits above the physical interface, so it holds only what the API gives it and what it believes: Table 8.

Table 8: The digital twin's parameters. The last two rows are its model of the pot: the same two symbols as the physical twin's model and, when the model is right, the same values. The pump constants a, b, a_{true} and b_{true} are absent on purpose; they live below the API, and the twin never sees them.

Symbol	Meaning	Unit
m_{t}	target moisture	per cent
k	gain: millilitres asked for per point of deficit	millilitres per percentage point
$V_{\max}$	the most the twin asks for in one dose	millilitres
T_{c}	time between the twin's checks	hours
d	the twin's belief about the pot's drying rate	percentage points per hour
g	the twin's belief about the pot's dose response	percentage points per millilitre

It decides. At each check the twin reads m and chooses a dose by the target-moisture rule of the Watering Strategies section, Equation 5: $V = \min\big(V_{\max},\ k\,(m_{\text{t}} - m)\big)$ when $m < m_{\text{t}}$, and no dose otherwise. Nothing is new here; the rule is recalled so that the model stands in one place.

It predicts. Because the twin carries the pot's two equations, it can say what the next reading should be before that reading arrives. Start from the reading m now, add what the dose should contribute by Equation 2, and subtract T_{c} hours of drying by Equation 1:

$$\hat{m} = m + g\,V - d\,T_{\text{c}} \tag{8}$$

$\hat{m}$ is the twin's expectation for the reading at the next check. Note what it is built from: the dose the twin *asked for*, not the volume delivered, because the twin cannot see the latter. That substitution is exactly the assumption a stale pump breaks.

It compares. When the next check arrives with the real reading m_{obs}, the twin computes the residual

$$r = \hat{m} - m_{\text{obs}} \tag{9}$$

A residual near zero, by Equation 9, says the model and the pot agree. A residual that stays positive says the pot is drier than the model expects at every check: either d is too small, or less water is arriving than was asked for. Which of the two it is can be read from *when* the shortfall appears – a wrong drying rate shows up as a larger drop *between* doses, a wrong pump as a smaller rise *at* them. That is the diagnosis the fixed schedule can never make, because it has no $\hat{m}$ to compare against.

It reports. The four metrics of Table 7 are the model's outputs rather than part of it. Equation 6 and W_{day} are read straight off the stored series, and Equation 7 is how the twin

recovers its own parameter d from the data – the one line in this chapter where the model is fitted rather than assumed.

Running the two together. One check, box by box. The twin reads m through a `GET`; it computes r from the $\hat{m}$ it wrote down last time; it chooses V by Equation 5 and sends it through a `PUT`. Below the API the controller applies Equation 3, the pump Equation 4, and the pot Equation 2 and then Equation 1 for T_c hours. The twin writes down the next $\hat{m}$ by Equation 8 and waits. Only two numbers cross the API in that loop, a reading going up and a dose coming down. Everything the rest of the book does to the twin – a d that depends on temperature and light, a fitted g, an uncertainty on $\hat{m}$, a rule for what to do when r leaves zero – is a change to one of these equations, not to the loop.

What the twin would be, so far

Put the pieces together and the shape of the first increment of a plant DT is visible before a line of it is written. It sits on the twin's side of the physical interface, speaking to the rig only through the REST API. Its physical adapter knows seven `GET`s and one `PUT`. It reads moisture, conductivity and watering events; it computes Equation 6, W_{day}, Equation 7 and the conductivity drift; it holds a target moisture that depends on what it believes the plant's state to be; it closes the loop by rewriting the schedule with Equation 5, under a cap; and it predicts the next reading with Equation 8 and checks it with Equation 9. That is *monitoring* for all four metrics, *control* for moisture, and, through the residual, *diagnosis* of the probe and the pump. Everything the rest of the book adds – the model's calibration, its uncertainty, the architecture that holds it – plugs into that shape.

Summary

Back to the objectives this chapter opened with.

1. **The block diagram.** Three groups – controller, sensor suite, actuators – and the pot they all end at. Four kinds of sensor (air temperature and humidity, light spectrum, and per pot either a capacitive moisture sensor or a three-quantity RS-485 probe), two kinds of actuator (a pump per pot and the relay channel that switches it), and the plumbing that is neither. Two buses: a short I2C chain for the sensors near the controller, a long RS-485 bus for the relay and the probes, and on each one every device has an address no other device on that wire shares.
2. **The four protocols.** I2C addresses chips on the desk and, in this rig, only reads. MODBUS RTU over RS-485 addresses devices across the room, reads registers and writes coils – the one place software touches the world. InfluxDB 3 turns readings into a time series named `<unit>_<parameter>`, kept for seven days. HTTP carries them off the desk as JSON. The only write towards the plant that an outsider can cause is one `PUT` of a schedule.
3. **The physical interface.** The REST API on port 8099 is the PT's software boundary; a DT depends on it and on nothing beneath it, because the layers beneath change, because responsibility changes hands there, and because the safety limit is enforced on the PT's side of it. In WLDT's vocabulary the DT-side counterpart is the physical adapter, with a shadowing function and a digital adapter behind it.
4. **Calibration.** Equation 3 turns a dose into a relay time with a per-pump slope and offset, fitted from a ladder of timed runs; the offset is tube and lift, the slope is the

reciprocal of the flow rate, both go stale, and the saved date is what lets a twin notice.

5. **Two strategies.** A fixed daily schedule leaves the loop open and the decision latency at days; a target-moisture rule, Equation 5, closes it through the API at a latency of one check interval, under a hard cap that is the safety argument moved up one level.
6. **Four metrics.** Time in the moisture band, Equation 6, is the value metric; water per day is its cost; the drying rate, Equation 7, is what a model must predict and what the residual is computed from; conductivity drift is the slow one a person acts on.
7. **Two models.** The physical twin on paper is four equations: the pot dries by Equation 1 and rises by Equation 2, the controller converts a dose with Equation 3 and its stored constants, and the pump delivers by Equation 4 and its real ones. The digital twin is the dose rule plus a prediction, Equation 8, and a residual, Equation 9; the residual is the one number that tells a stale pump from a hot day.

The one sentence to carry forward: **the twin sees the plant only through the API, and everything below that line is the physical twin's business.**

Exercises

Solutions or hints follow the exercises; each says which. The objective each exercise serves is given in brackets.

Exercise 1 (Objective 2, easy). A reading of soil temperature arrives in a browser as `{"value": 21.4, "units": "°C"}`. Name every protocol it passed through, in order, and for each say what the datum looked like at that hop – a register value, a Python float, a point, a JSON object – and one thing that hop added which the previous one did not have.

Exercise 2 (Objective 4, medium). Finish Example 2: compute the relay time for $V = 80$ using the fitted constants, then redo the dose and the relay time for $m = 21$, and name the number that limited the second dose.

Exercise 3 (Objective 4, medium). The calibration in Example 1 is repeated with the outlet 60 cm above the water instead of 30 cm. Say, with a reason, whether you expect a to change, whether you expect b to change, and in which direction each moves. Then say what a dose of 20 ml would deliver under the *old* constants if the geometry had changed and nobody recalibrated.

Exercise 4 (Objective 5, medium). Write, as a sequence of HTTP requests with bodies, one full cycle of the target-moisture strategy for a plant named `plant_2` with $m_t = 35$, $k = 10$ and $V_{\max} = 150$, starting from a reading of $m = 30$. Include the request that confirms the dose was delivered and the request that leaves the plant quiet until the next check.

Exercise 5 (Objective 3, medium). A colleague proposes that the DT should read the soil probe's MODBUS registers directly over a second RS-485 adapter, "to avoid the latency of the API". Write the three- sentence reply you would give, using the three reasons in the Physical Interface section, and say which one of the three is the one that should end the discussion.

Exercise 6 (Objective 6, medium). For each of the four metrics in Table 7, state the single `GET` request (or pair of requests) that fetches the data needed to compute it for one day, and say which metric cannot be computed for a plant fitted with the cheaper STEMMA sensor and why.

Exercise 7 (Objectives 4 and 5, hard). Do Example 3, the stale calibration, in full. State how each dose changes under both strategies when the lift grows and the tube lengthens; say which strategy's stored data shows the problem first and in which series; and list what a DT would have to have recorded, beyond the rig's own data, to raise the alarm rather than merely to have made it possible in hindsight.

Exercise 8 (Objective 2, hard, open-ended). The database keeps seven days of history. Argue for and against raising it to a year on the Raspberry Pi, and then design the smallest change – on either side of the physical interface – that would give a DT a year of history without changing the retention setting. There is no single correct answer; there is a correct form of answer, which says where the copy lives and who owns keeping it correct.

Solutions and hints

Exercise 1 (solution). RS-485 / MODBUS RTU: a sixteen-bit holding register holding `214`, read at the probe's device id – the hop adds an address on a shared cable. Controller software: the Python float `21.4` after division by ten, wrapped in a measurement with units and a timestamp – the hop adds units and time. InfluxDB 3: a point in `plant_1_soil_temperature` with tags and fields – the hop adds a name, an index and persistence. HTTP: a JSON object in an array returned by

```
GET /sensing/plant_1/soil_temperature
```

– the hop adds reachability from anywhere on the network and a status code that says whether the request made sense.

Exercise 2 (solution). $t = 0.0197 \times 80 + 1.56 = 3.14$ seconds. At $m = 21$ the deficit is nineteen points and the rule asks for 190 ml, which the cap reduces to $V_{\max} = 150$; the relay time is then $0.0197 \times 150 + 1.56 = 4.52$ seconds. The cap limited the dose, not the gain.

Exercise 3 (hint). The slope is about flow once water is moving, and flow through a submersible pump falls only slowly with lift, so expect a to rise a little. The offset is about filling the tube and lifting the first water, so expect b to rise clearly. Under the old constants a 20 ml dose runs the pump for $t = 0.0197 \times 20 + 1.56 = 1.95$ seconds; if the new offset is above that, the dose delivers nothing at all.

Exercise 4 (hint). Deficit five points, dose 50 ml. Four requests, in this order:

```
GET  /sensing/plant_2/soil_moisture?limit=1
PUT  /actuation/plant_2/update_schedule      (one event, a minute ahead)
GET  /actuation/plant_2/watering_events?limit=1   (repeat until it
 ↪ appears)
PUT  /actuation/plant_2/update_schedule      (an empty schedule list)
```

Say what your program does if the third request never returns a new event.

Exercise 5 (hint). The reason that ends the discussion is the third: below the line, two clients on one MODBUS bus with no shared lock will corrupt each other's frames, and the API's inability to switch a coil is the rig's safety property. Latency is the wrong worry; the probe is read every ten seconds regardless of who asks.

Exercise 6 (hint). Moisture and conductivity come from `/sensing/{unit}/{parameter}` with `since_timestamp` set to midnight; water per day from `/actuation/{unit}/`

`watering_events` the same way; the drying rate needs both series to find the readings around each dose. Conductivity drift needs the RS-485 probe; a STEMMA-only plant has no conductivity series to compute it from.

Exercise 7 (hint). Both strategies under-deliver by the same millilitres per dose, so the fixed schedule shows nothing in its own data – its watering events still say 100 and 120 – and the problem is visible only in the moisture series trending down. The target-moisture rule compensates by dosing more often, so its watering-event count rises while the moisture stays near target; the tell is the rising W_{day}. To raise the alarm rather than explain it afterwards, a DT needs the calibration constants and their date, and an expected drying rate to compare against.

Exercise 8 (open). Assessed on whether the answer names the memory card's size and the write load on one side and the DT's own store and its ownership on the other, and on whether the proposed copy runs on the twin's side of the physical interface.

Further Reading

- *Building the rig:* the plant-controller repository holds the build documentation this chapter deliberately leaves out – parts list, assembly guide, software installation and the REST API reference – for a reader who wants a rig of their own.
- *Comparable small twins of plants:* [1] for the GreenhouseDT exemplar and its watering-pump control, and [14] for why a target moisture should depend on the plant's stage; both are the closest published relatives of this chapter's rig.
- *Sensing, communication and storage for twins:* [3] on sampling frequency, communication delay and time-series storage; [6] and [7] for the incubator case study's use of a message broker and InfluxDB, all three in [17].
- *Protocols:* [8] weighs four common IoT messaging protocols – message queuing telemetry transport (MQTT), CoAP, advanced message queuing protocol (AMQP) and HTTP; [4] tallies which protocols predictive-maintenance twins actually use, MODBUS among them; [5] is the IoT reference architecture that places MODBUS among data-transferring capabilities.
- *The physical adapter idea:* [10] introduces the WLDT library, [12] generalises it into an edge digital twin with shadowing, [13] shows the physical adapter, shadowing function and digital adapter in an industrial case, [9] treats PT and DT lifecycles, and [11] surveys WLDT among other IoT twin platforms.
- *Agricultural twins at field scale:* [15] for an irrigation-pivot twin on an open platform, [16] for architecture patterns illustrated on agriculture and food cases, and the smart-farming section of [11].
- *Reproducible research rigs of the same family:* [2] on reproducible twin research with a small robot car, and on ocean platforms whose sensors and actuators sit on an RS-485 fieldbus behind a Raspberry Pi.
- *Consulted but not drawn on above,* and worth knowing they exist: [18] for a platform that also settled on InfluxDB for its time series, and [19] for CoAP, the constrained-device cousin of HTTP that [8] compares it with.

References

[1] E. Kamburjan *et al.*, "GreenhouseDT: An Exemplar for Digital Twins," in *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, in SEAMS '24. New York, NY, USA: Association for Computing Machinery, Jun. 2024, pp. 175–181. doi: 10.1145/3643915.3644108.

[2] A. Barbie and W. Hasselbring, "Toward Reproducibility of Digital Twin Research: Exemplified with the PiCar-X." arXiv, Aug. 2024. doi: 10.48550/arXiv.2408.13866.

[3] C. Gomes, D. E. L. Rötter, A. Iosifidis, H. Feng, H. Ejersbo, and M. Frasheri, "Sensing and Communication of Data from the Physical Twin," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds., Cham: Springer International Publishing, 2024, pp. 147–171. doi: 10.1007/978-3-031-66719-0_7.

[4] R. van Dinter, B. Tekinerdogan, and C. Catal, "Predictive maintenance using digital twins: A systematic literature review," *Information and Software Technology*, vol. 151, p. 107008, Nov. 2022, doi: 10.1016/j.infsof.2022.107008.

[5] ISO, "ISO/IEC 30141: Internet of things (IoT) – Reference architecture."

[6] C. Gomes, B. J. Oakes, J. Fitzgerald, and P. G. Larsen, "Foundational Concepts for Digital Twins of Cyber-Physical Systems," in *The Engineering of Digital Twins*, J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds., Cham: Springer International Publishing, 2024, pp. 45–63. doi: 10.1007/978-3-031-66719-0_3.

[7] C. Gomes *et al.*, "Digital Twin Tutorial: The Incubator Case Study," in *Engineering Trustworthy Software Systems: 6th International School, SETSS 2024, Chongqing, China, April 14–21, 2024, Tutorial Lectures*, J. P. Bowen, C. Gomes, and Z. Liu, Eds., Singapore: Springer Nature, 2025, pp. 68–101. doi: 10.1007/978-981-96-4656-2_3.

[8] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in *2017 IEEE international systems engineering symposium (ISSE)*, IEEE, 2017, pp. 1–7. Accessed: Feb. 25, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8088251/

[9] M. Picone *et al.*, "Harmonizing Physical and Digital Twins Lifecycles," in *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*, Mar. 2025, pp. 197–204. doi: 10.1109/ICSA-C65153.2025.00039.

[10] M. Picone, M. Mamei, and F. Zambonelli, "WLDT: A general purpose library to build IoT digital twins," *SoftwareX*, vol. 13, p. 100661, Jan. 2021, doi: 10.1016/j.softx.2021.100661.

[11] A. Hakiri, A. Gokhale, S. B. Yahia, and N. Mellouli, "A comprehensive survey on digital twin for future networks and emerging Internet of Things industry," *Computer Networks*, vol. 244, p. 110350, May 2024, doi: 10.1016/j.comnet.2024.110350.

[12] M. Picone, M. Mamei, and F. Zambonelli, "A Flexible and Modular Architecture for Edge Digital Twin: Implementation and Evaluation," *ACM Transactions on Internet of Things*, vol. 4, no. 1, pp. 8:1–8:32, Feb. 2023, doi: 10.1145/3573206.

[13] M. Martinelli, J. Zhang, A.-K. Splettstoßer, M. Picone, M. Lippi, and A. Wortmann, "Hierarchical Digital Twin Ecosystem for Industrial Manufacturing Scenarios," in *2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2024, pp. 56–63. doi: 10.1109/SEAA64295.2024.00018.

[14] E. Kamburjan, N. Bencomo, S. L. Tapia Tarifa, and E. B. Johnsen, "Declarative Lifecycle Management in Digital Twins," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, Linz Austria: ACM, Sep. 2024, pp. 353–363. doi: 10.1145/3652620.3688248.

[15] S. Infante *et al.*, “Integrating FMI and ML/AI models on the open-source digital twin framework OpenTwins,” *Software Practice and Experience*, Mar. 2024, doi: 10.1002/spe.3322.

[16] B. Tekinerdogan and C. Verdouw, “Systems Architecture Design Pattern Catalog for Developing Digital Twins,” *Sensors*, vol. 20, no. 18, p. 5103, Jan. 2020, doi: 10.3390/s20185103.

[17] J. Fitzgerald, C. Gomes, and P. G. Larsen, Eds., *The Engineering of Digital Twins*. Cham: Springer International Publishing, 2024. doi: 10.1007/978-3-031-66719-0.

[18] N. Li *et al.*, “Six-dimensional digital twin modeling and software platform design for complex industrial systems,” *Journal of Intelligent Manufacturing*, Jan. 2025, doi: 10.1007/s10845-025-02567-8.

[19] M. Kovatsch, M. Lanter, and Z. Shelby, “Californium: Scalable cloud services for the internet of things with coap,” in *2014 International Conference on the Internet of Things (IOT)*, IEEE, 2014, pp. 1–6. Accessed: Sep. 10, 2024. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7030106/