

# 0007 — Report: services.py — RCT and Decision-Making Services

## 1. Purpose and role

The module `dtwinpy/services.py` implements the **Service\_Handler** class: external services of the Digital Twin, in particular:

- **RCT (Remaining Cycle Time) prediction** — Simulate until a given part is finished and return its cycle time.
- **RCT-based decision making at branch points** — For parts approaching a branching machine (one with multiple output queues), generate all **path scenarios** (combinations of branch choices), simulate each scenario, compare RCTs, and **publish** the best path to the physical system (e.g. via MQTT) when the gain is above a threshold.

**Branch point:** A machine with **multiple output queues** (and thus multiple conveyors). The physical system must choose where to send each part; the service recommends the path with the **lowest RCT** when the improvement is significant.

---

## 2. Dependencies and imports

```
from .broker_manager import Broker_Manager
from .interfaceDB import Database
from .helper import Helper
from matplotlib import pyplot as plt
import re
from time import sleep
```

- **Broker\_Manager** — MQTT (or similar) publishing of the chosen path to the physical system.
  - **Database** — ID database (branch queue selections, PalletID ↔ PID) and experiment database (RCT tracking).
  - **Helper** — Logging and time.
  - **re** — Extract part ID from part name in **publish\_feedback** and **return\_first\_branch**.
  - **sleep** — Delay between MQTT messages in **publish\_feedback**.
- 

## 3. Service\_Handler class — overview

Single class **Service\_Handler**. Main flow:

- **Constructor:** Build digital model via **generate\_digital\_model()**, create ID and exp databases, get branches and components, set **rct\_threshold** and **queue\_position**.
  - **run\_RCT\_service()** — Full RCT decision-making pipeline: generate path scenarios → get parts making decisions → simulate all paths per part → RCT\_check (gain vs threshold) → **publish\_feedback** (MQTT) → return first-branch results.
  - **run\_RCT\_tracking()** — For a given **palletID**: resolve **PID**, run a single simulation targeted at that part, compute RCT, write to exp database.
- 

## 4. Constructor and main attributes

```
def __init__(self, name, generate_digital_model, broker_manager, rct_threshold=0.02,
             queue_position=2, flag_publish=True):
```

Parameter / attribute	Description
<b>name</b>	Service name.
<b>generate_digital_model</b>	Callable that returns a <b>Model</b> (e.g. with optional args like verbose, targeted_part_id).
<b>broker_manager</b>	Broker_Manager instance for MQTT publishing; can be None (no publish).
<b>rct_threshold</b>	Minimum gain (e.g. 0.02 = 2%) above which the best path is recommended.
<b>queue_position</b>	1-based position in the branch input queue that triggers “part making decision” (default 2 = second part in queue).
<b>flag_publish</b>	If True, <b>publish_feedback</b> sends MQTT messages; if False, only computes and does not publish.
<b>digital_model</b>	Result of <b>generate_digital_model(verbose=False)</b> .
<b>ID_database</b>	Database at path derived from digital DB path (replace “digital” with “ID”), event_table “ID” — stores branch queue selections, PalletID ↔ PID.
<b>exp_database</b>	Database at path with “exp” — stores RCT tracking (PID, RCT, palletID).
<b>branches</b>	From <b>digital_model.get_branches()</b> .
<b>machines_vector, queues_vector</b>	From <b>digital_model.get_model_components()</b> .
<b>part_vector</b>	From <b>digital_model.get_all_parts()</b> .

## 5. Branch choices and path scenarios

### 5.1 get\_branch\_choices()

- For each **branch** in **branches**, get **branch.get\_conveyors()** (list of conveyors = output choices).
- Returns **list of lists**: [ [conveyors of branch 1], [conveyors of branch 2], ... ].

Used to build all possible **paths** as combinations of one conveyor per branch.

### 5.2 generate\_path\_scenarios(verbose=False)

- Calls **get\_branch\_choices()**.

- Builds **all combinations**: one conveyor per branch per combination (recursive helper).
- Returns **path\_scenarios**: list of **paths**, each path = list of **Conveyor** objects (one per branch, in branch order).
- If **verbose**, prints each path with conveyor names.

So **path\_scenarios** has size = product of branch sizes (e.g. 2 branches with 2 choices each → 4 paths).

### 5.3 get\_parts\_making\_decisions(queue\_position=2)

- For each branch, get **branch.get\_branch\_queue\_in()** (input queue(s)).
- For each queue, get **queue.get\_all\_items()**.
- If **len(parts\_in\_queue) > queue\_position - 1**, the part at index **queue\_position - 1** (1-based “second position” = index 1) is a “part making decision”; append it.
- If **any** branch queue has **no** part at that position, returns **False** (no decision to make).
- Otherwise returns **list of parts** (one per branch, the part at the chosen queue position).

So the service only runs when **every** branch has at least one part in the required position.

## 6. Step-by-step: how the simulation and scenarios work

### 6.1 What is a “path scenario”?

- A **branch point** is a machine with **several output queues** (e.g. Machine 2 → Queue to Machine 3 or Queue to Machine 4).
- A **path** is one **combination of choices** at every branch: e.g. “at branch 1 choose conveyor A, at branch 2 choose conveyor C”.
- **Path scenarios** = **all such combinations**. Example: 2 branches with 2 options each → 4 paths (A1, A2, B1, B2).

Each path is represented as a **list of Conveyor objects** (one per branch, in branch order). The machine uses **part.branching\_path** to decide which conveyor (and thus which queue) to use when the part leaves.

### 6.2 Why we run one simulation per scenario

- The **physical system** will send the part through **one** of the possible paths. We do **not** know in advance which path will be chosen.
- To **recommend** the best path, we run the **digital model once per path** (plus once “AS IS” with current policy). For each run we **stop when the part of interest finishes** and record its **RCT (Remaining Cycle Time)**.
- **Lower RCT** = that path gets the part finished sooner. We compare RCTs and recommend the path with the **lowest RCT**, but only if the **gain** (vs current “AS IS”) is above **rct\_threshold**.

### 6.3 Step-by-step flow of simulate\_paths

Step	Who	Action
1	Service	Take <b>parts_making_decisions</b> (one part per branch at <b>queue_position</b> , e.g. second in queue).
2	Service	For <b>each part</b> in that list (e.g. Part 5):
3	Service	<b>Generate a fresh model</b> with <b>targeted_part_id = part.get_id()</b> so the simulation stops when this part completes.
4	Service	Load <b>parts_branch_queue</b> from ID DB and set on all machines (current policy).
5	Model	<b>Run simulation</b> with no extra path override (current alternated/RCT policy).
6	Model	Simulation stops when <b>Part 5</b> finishes at the final machine.
7	Service	<b>calculate_RCT(part_id_selected=5) → rct_asis</b> ; append to <b>rct_vector</b> .
8	Service	For <b>each path</b> in <b>possible_paths</b> (e.g. Path 0, Path 1, Path 2):
9	Service	<b>Generate a fresh model</b> again (same targeted_part_id).
10	Service	Set <b>parts_branch_queue</b> again.
11	Service	In the new model, find the <b>Part 5</b> object and call <b>part.set_branching_path(path_scenario)</b> so that at branch points this part <b>always</b> follows this path.
12	Model	<b>Run simulation</b> ; Part 5 is forced along the chosen path.
13	Service	<b>calculate_RCT(part_id_selected=5) → rct_path_k</b> ; append to <b>rct_vector</b> .
14	Service	After all paths for Part 5: <b>rct_dict["Part 5"] = [rct_asis, rct_path0, rct_path1, ...]</b> .
15	Service	Repeat from step 2 for the next part (if any).
16	Service	Clear <b>targeted_part_id</b> and <b>targeted_cluster</b> so the model is not left in "stop at part" mode.

So for **P** parts and **S** path scenarios, we run **P × (1 + S)** simulations (each with a new model and one run).

## 6.4 What about the part behind (and all other parts)?

When we simulate “Part A goes to Machine 3” vs “Part A goes to Machine 4”, we only **force the path for the part we are evaluating** (the one at **queue\_position**). **All other parts** — including the part **ahead** of Part A in the queue and the part **behind** Part A — **do not** get a path set in that run.

- In code, only the part with `current_part.get_id() == part.get_id()` gets `current_part.set_branching_path(path_scenario)`. No other part receives `set_branching_path` in that scenario.
- So for every other part, the **machine** decides the path using its **default** behaviour:
  - If the part is in **parts\_branch\_queue** (from the ID database), the machine uses the **RCT** policy (recommended queue from previous recommendations).
  - Otherwise the machine uses its **alternated** policy (e.g. round-robin).

So we **do not** simulate all combinations of “Part A path × Part B path × ...”. We **fix** the part behind (and everyone else) to the **current/default** policy (ID DB + alternated) and only **vary** the path for the single part we are evaluating. That avoids a combinatorial explosion and an infinite cascade of “what if the part behind goes to 3 or 4?”. The answer the code gives is: *“If this part takes path 3, and everyone else behaves as they do today (current policy), what is this part’s RCT?”*

---

## 7. Assigning paths and simulating (reference)

### 7.1 assign\_parts(SelecPath, path\_scenarios, SelecPart=None)

- **selected\_path** = `path_scenarios[SelecPath]`.
- **Approach 1 (SelecPart == None)**: For each branch, if the branch input queue has more than one part, take the part at **index 1** (second position), call **part.set\_branching\_path(selected\_path)**, and append to **parts\_in\_branching\_dm**. Same path is assigned to all such parts at that branch.
- **Approach 2 (SelecPart != None)**: Only the part with **part.get\_id() == SelecPart.get\_id()** gets **set\_branching\_path(selected\_path)** (other parts keep default).
- Returns **parts\_in\_branching\_dm** (list of parts that received the path; used for Approach 1).

**simulate\_paths** does not use **assign\_parts** directly; it regenerates the model and assigns the path by part ID for each scenario.

### 7.2 simulate\_paths(possible\_pathes, parts\_making\_decisions, verbose=True, plot=False)

Implements the flow described in **Section 6.3**: for each part, one AS IS run then one run per path; each run uses a newly generated model and **targeted\_part\_id** so the simulation stops when that part finishes. **rct\_dict** stores, per part, the list **[rct\_asis, rct\_path1, rct\_path2, ...]**.

Optional **plot**: bar chart of RCT per path per part; boxplot of max/min RCT per part.

---

## 8. RCT check and feedback

### 8.1 RCT\_check(rct\_dict, rct\_threshold, possible\_pathes, verbose=True, plot=False)

For **each part** in **rct\_dict**:

- **rcts\_paths** = `rct_dict[key]`; **ASIS\_RCT** = `rcts_paths[0]`.
- Remove first element (AS IS).

- For each remaining **rct**: **rct\_indicator** = rct / ASIS\_RCT; **rct\_gain** = 1 - rct\_indicator; append to **gain\_vect**.
- **highest\_gain** = max(gain\_vect).
- If **highest\_gain** >= **rct\_threshold**: **flag\_feedback** = True, **path\_to\_implement** = index of that gain in gain\_vect (0-based path index).
- **feedback\_dict[key]** = (flag\_feedback, path\_to\_implement, highest\_gain).

So **path\_to\_implement** is the index in **possible\_pathes** of the best path for that part; if no path beats the threshold, flag is False and index is 0.

Optional **plot**: gain per path per part, with threshold line. **Return**: **feedback\_dict**.

## 8.2 publish\_feedback(feedback\_dict, possible\_pathes)

For **each part** in feedback\_dict:

- Parse **part\_id** from part name (e.g. "Part 3" → 3).
- **feedback\_flag**, **path\_to\_implement**, **gain** = feedback\_dict[part\_name].
- **selected\_path** = possible\_pathes[path\_to\_implement].

If **feedback\_flag** is True:

- Find **part\_location** from **part\_vector** (part.get\_location()).
- Find the **branch** whose input queue id matches **part\_location** (queue id - 1 == part\_location) to get **machine\_selected** (branch machine) and **selected\_branch\_id**.
- **queue\_id** = selected\_path[selected\_branch\_id - 1].id (conveyor id = queue id).
- If **broker\_manager** is not None and **flag\_publish** is True: **broker\_manager.publishing(machine\_id, part\_id, queue\_id, topic="RCT\_server")**.
- Append **queue\_id** and **gain** to **queues\_selected** and **gains**.
- **sleep(1)** before next message.

**Return**: (feedback\_flag, queues\_selected, gains). Note: if multiple parts have feedback, only one **feedback\_flag** is returned (last in loop); **queues\_selected** and **gains** list all.

## 8.3 return\_first\_branch(rct\_dict, publish\_results)

- Takes the **first** part name in **rct\_dict**.
- **part\_id** from name; **path\_1** = rct\_dict[part\_name][0] (AS IS RCT); **path\_2** = rct\_dict[part\_name][1] (first path RCT).
- **feedback\_flag** = publish\_results[0]; **queue\_id** = publish\_results[1]; **gains** = publish\_results[2].
- **Return**: (**part\_id**, **path\_1**, **path\_2**, **queue\_id**, **feedback\_flag**, **gains**) — used for API or main Digital Twin code.

# 9. Main entry points

## 9.1 run\_RCT\_service(queue\_position=2, verbose=True, plot=False)

1. **possible\_pathes** = generate\_path\_scenarios(verbose).
2. **parts\_making\_decisions** = get\_parts\_making\_decisions(queue\_position).
3. If **parts\_making\_decisions** is False → log warning and **return False** (no parts in position).
4. **rct\_dict** = simulate\_paths(possible\_pathes, parts\_making\_decisions, verbose, plot).
5. **feedback\_dict** = RCT\_check(rct\_dict, rct\_threshold, possible\_pathes, verbose, plot).
6. **publish\_results** = publish\_feedback(feedback\_dict, possible\_pathes).
7. **rct\_results** = return\_first\_branch(rct\_dict, publish\_results).

8. **return rct\_results** (tuple: part\_id, rct\_asis, rct\_path1, queue\_id, feedback\_flag, gains).

So one call runs the full pipeline: path generation → part selection → simulations → RCT comparison → optional MQTT publish → first-branch summary.

## 9.2 run\_RCT\_tracking(palletID)

- **PID** = **ID\_database.get\_PID\_from\_PalletID(palletID)** (translate pallet to part ID).
- **targeted\_PID** = integer from PID.
- **digital\_model** = **generate\_digital\_model(targeted\_part\_id=targeted\_PID)**.
- **part\_in\_model** = **digital\_model.check\_partID\_in\_simulation(PID)**.
- If **True**: **digital\_model.run()** → **RCT** = **calculate\_RCT(part\_id\_selected=targeted\_PID)** → **exp\_database.write\_RCTtracking(PID, RCT, palletID)** and log.
- If **False**: log warning (part not in model, e.g. pallet not yet at Machine 1).

Used to track RCT for a specific pallet over time; results are stored in the experiment database.

---

## 10. Sequence diagram — RCT service (simulate\_paths and run\_RCT\_service)

The following sequence diagram shows how **run\_RCT\_service** and **simulate\_paths** interact with the digital model and how each scenario is run for one part.

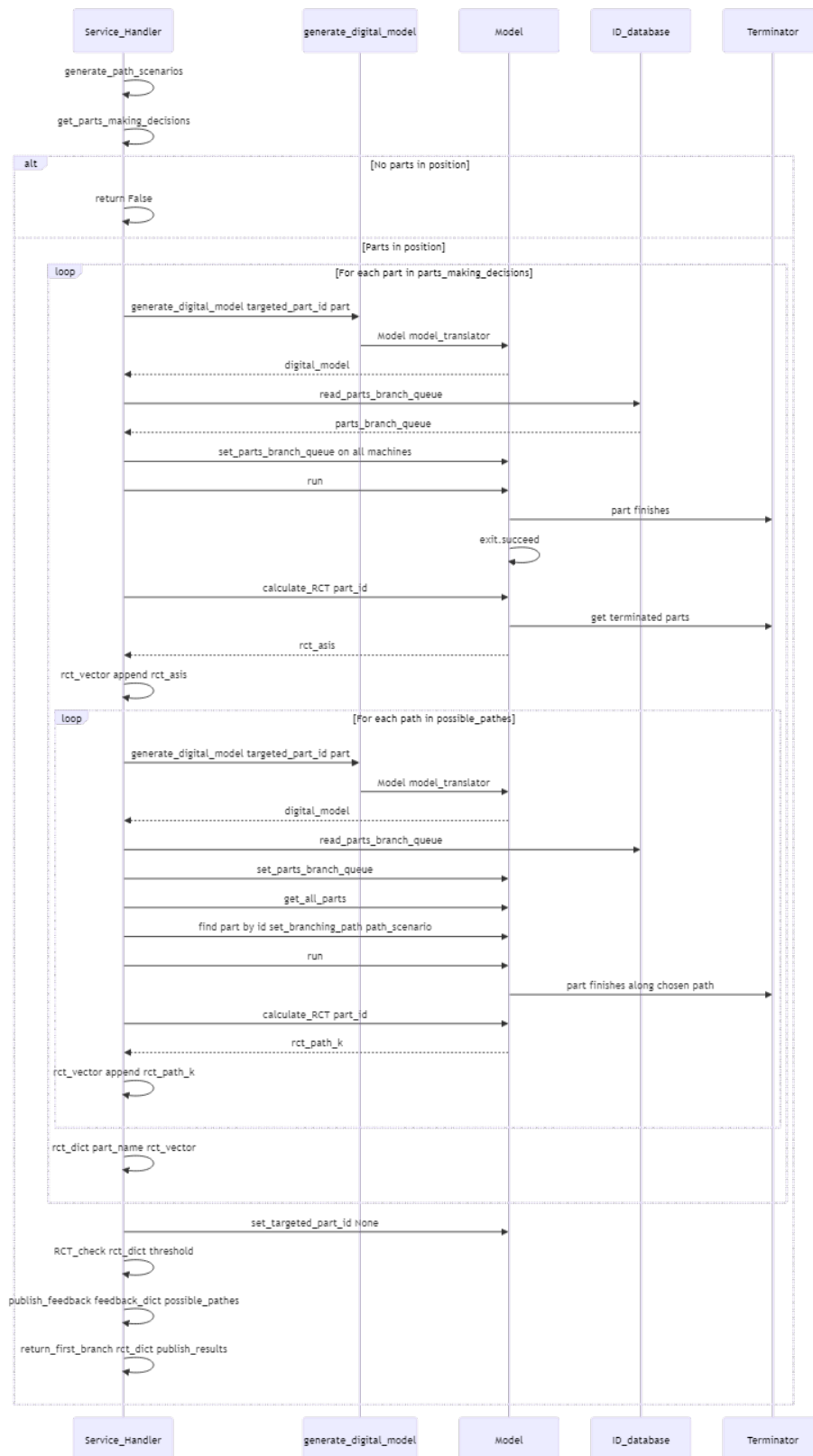
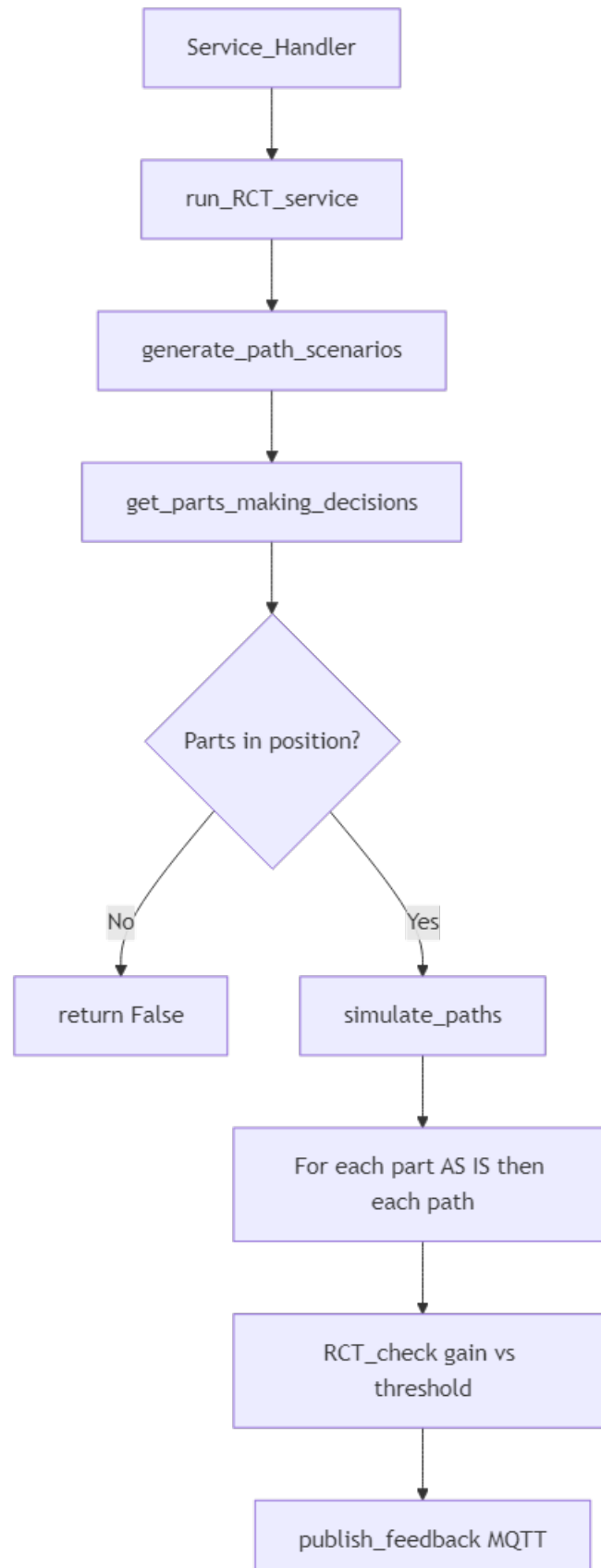


Diagram 1



## 11. High-level flow (Mermaid)



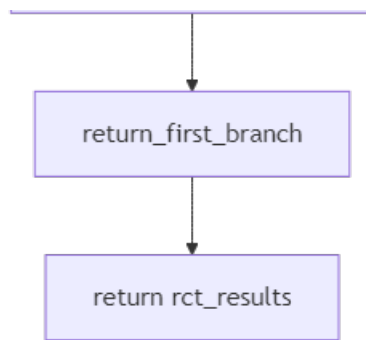


Diagram 2

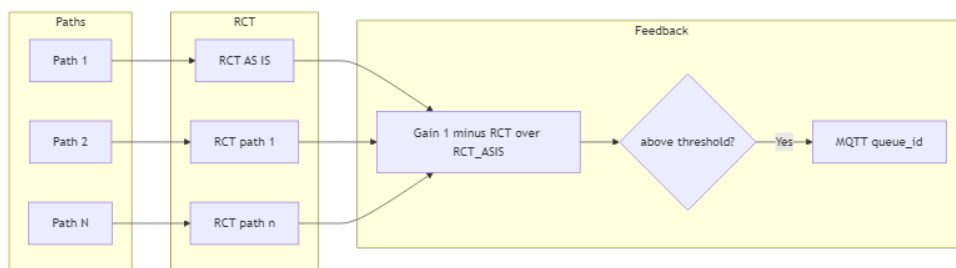


Diagram 3

## 12. Data structures (summary)

Structure	Description
<b>path_scenarios</b>	List of paths; each path = list of Conveyor (one per branch).
<b>rct_dict</b>	{ "Part N": [rct_asis, rct_path1, rct_path2, ... ] }.
<b>feedback_dict</b>	{ "Part N": (flag_feedback, path_to_implement, highest_gain) }.
<b>publish_results</b>	(feedback_flag, queues_selected, gains).
<b>rct_results</b>	(part_id, path_1_rct, path_2_rct, queue_id, feedback_flag, gains) for first part.

## 13. Relation to other modules

Module / concept	Relation to services
<b>digital_model</b>	Built via <b>generate_digital_model()</b> ; used for get_branches, get_model_components, get_all_parts, run, calculate_RCT, check_partID_in_simulation, set_targeted_part_id/cluster. Model is <b>regenerated</b> per scenario and per part in simulate_paths.
<b>broker_manager</b>	<b>publishing(machine_id, part_id, queue_id, topic="RCT_server")</b> to send chosen queue to the physical system.
<b>interfaceDB</b>	<b>ID_database</b> : read_parts_branch_queue, get_PID_from_PalletID. <b>exp_database</b> : write_RCTtracking.
<b>components</b>	<b>Part</b> : set_branching_path, get_id, get_location. <b>Branch</b> : get_conveyors, get_branch_queue_in, get_branch_machine, get_id. <b>Machine</b> : set_parts_branch_queue.

## 14. Summary

- **services.py** provides **Service\_Handler** for **RCT prediction** and **RCT-based decision making** at branch points.
- **Path scenarios** = all combinations of one conveyor per branch; **parts making decisions** = parts at a given queue position (e.g. second) in each branch input queue.
- **simulate\_paths** runs, for each such part, one "AS IS" simulation plus one per path (model regenerated each time, targeted\_part\_id set); **rct\_dict** stores RCT per part per scenario.
- **RCT\_check** computes gain ( $1 - \text{RCT}/\text{RCT\_ASIS}$ ) per path; if max gain  $\geq$  **rct\_threshold**, **feedback\_dict** marks the best path and **publish\_feedback** sends it via **broker\_manager** (machine\_id, part\_id, queue\_id).
- **run\_RCT\_service** runs the full pipeline and returns **return\_first\_branch** (part\_id, rct\_asis, rct\_first\_path, queue\_id, feedback\_flag, gains).
- **run\_RCT\_tracking** computes RCT for a given **palletID** (via PID) and writes PID, RCT, palletID to **exp\_database**.