

Build a physical AI data factory with NVIDIA Cosmos

An inspection model is only as good as the defects it has seen, and the defects that matter most are the rarest. This guide shows how to build a loop that generates them: NVIDIA Cosmos paints realistic defects onto real boards, a quality gate keeps only the ones that measure up, and a coding agent drives the whole cycle on Klique from a goal written in plain English.

Rare defects, on demand

Generate the defect images a production line seldom yields

NVIDIA's stack, as shipped

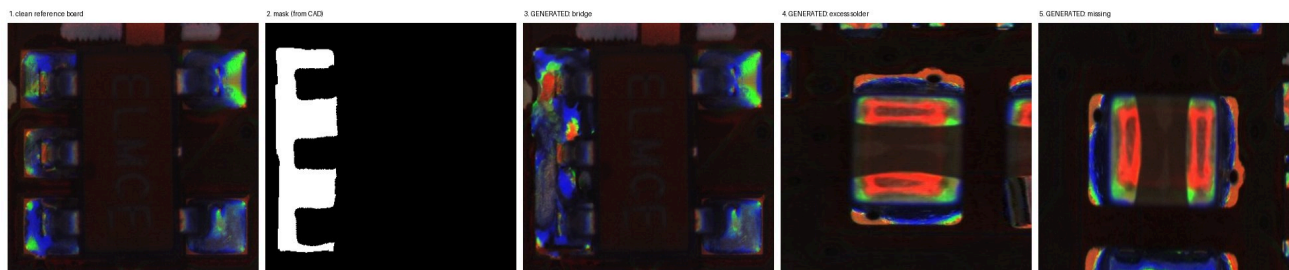
Cosmos models and containers run unmodified

Agent-driven

A coding agent runs the loop from a goal and a budget

Repeatable

Any stage can be reopened, cloned and run again



These defects were never photographed. Left to right: a real, clean board; the mask placed from the board's own CAD; then a solder bridge, excess solder and a missing component, generated by NVIDIA Cosmos AnomalyGen as tracked tasks on Klique.

01

Rare defects are the hardest to train for

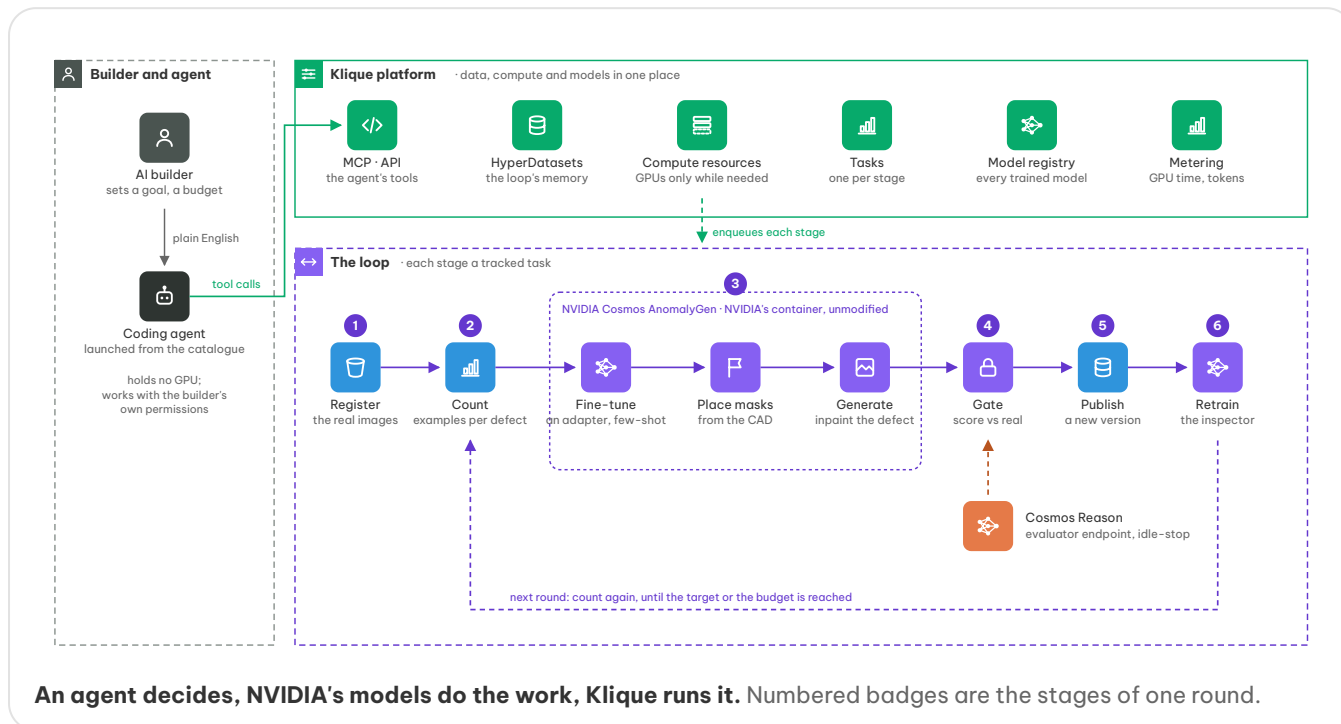
Consider this: a manufacturing quality team is judged on one number: how many bad boards get through when every defect must be caught. The hard part is not the model. Serious faults are rare because the process works, so the training set is thinnest exactly where a miss costs most. A typical inspection set holds plenty of examples of a common fault, such as a missing component, and only a handful of a solder bridge, the fault that causes a short.

Collecting more is not an option; the line would have to fail more often. NVIDIA's answer is to **generate** the missing defects with [Cosmos](#), its open world foundation model, and to do it in a loop that keeps improving the inspection model: the [Physical AI Data Factory blueprint](#). NVIDIA ships the models and the method, including the open-source [Cosmos AnomalyGen](#) defect generator. It leaves the infrastructure to the builder: GPUs on demand, a home for the data and its versions, and a way to run each stage. That is the part Klique supplies.

Goal: every defect class at a target number of trustworthy examples, and an inspection model retrained on them, in a loop that can be run again whenever new defects appear.

Blueprint at a glance

Here is how we put it together. An **agent** makes the decisions. **NVIDIA's models** do the heavy work, unmodified. **Klique** supplies the GPUs, the dataset versions and the model registry, and runs every stage as a task on a GPU that is held only while needed.



THE AGENT

Decides

Whether a round is worth running

How many of each defect to request

Whether a result is worth improving

When to stop

NVIDIA'S MODELS

Do the work

Cosmos AnomalyGen paints the defect

Automatic Mask Placement finds where

nn_score measures it against real data

Cosmos Reason checks plausibility

KLIQUE

Runs it

A GPU for each stage, only while needed

A new dataset version every round

A registry for every trained model

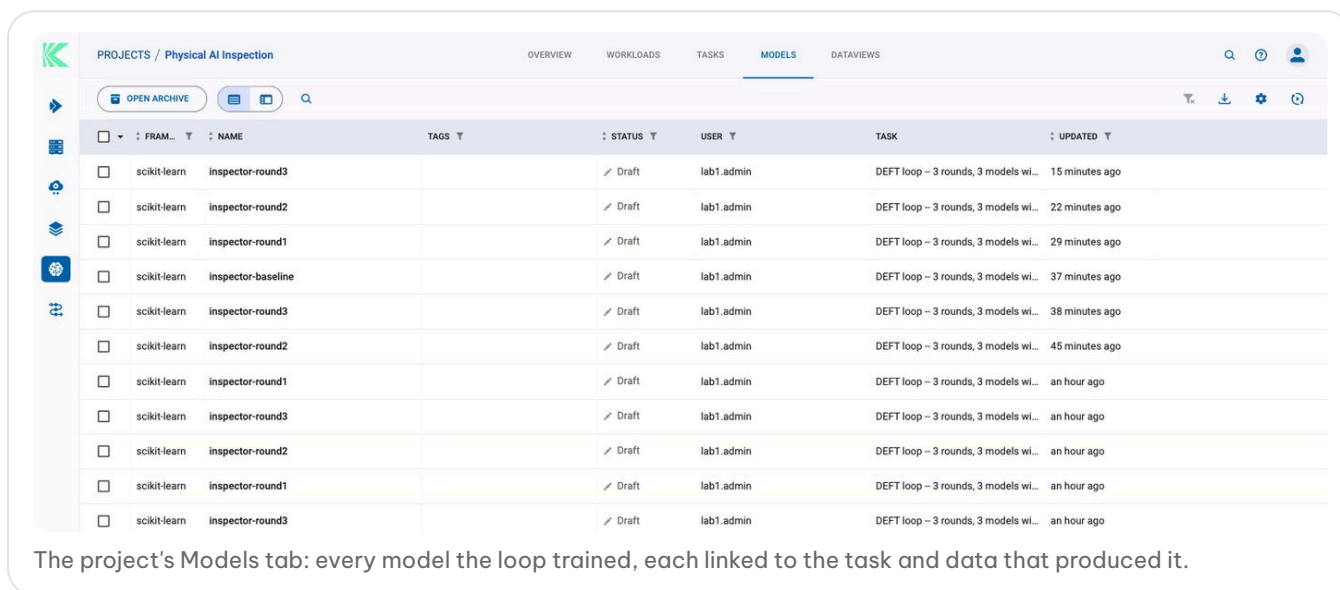
Quotas and access applied to every action

How it runs on Klique

Before the stages, a short tour of the platform pieces this build leans on. A builder new to Klique needs only a few ideas.

Projects: one home for the whole build

Everything in this guide lives in one project, *Physical AI Inspection*. Its tabs hold every task the loop ran, every pipeline run, every dataset version and every trained model, so the whole build can be found, compared and shared from one place. Projects nest like folders, which keeps one team's experiments apart from another's, and access is granted per project.



The project's Models tab: every model the loop trained, each linked to the task and data that produced it.

FRAM...	NAME	TAGS	STATUS	USER	TASK	UPDATED
scikit-learn	inspector-round3		Draft	lab1.admin	DEFT loop - 3 rounds, 3 models wi...	15 minutes ago
scikit-learn	inspector-round2		Draft	lab1.admin	DEFT loop - 3 rounds, 3 models wi...	22 minutes ago
scikit-learn	inspector-round1		Draft	lab1.admin	DEFT loop - 3 rounds, 3 models wi...	29 minutes ago
scikit-learn	inspector-baseline		Draft	lab1.admin	DEFT loop - 3 rounds, 3 models wi...	37 minutes ago
scikit-learn	inspector-round3		Draft	lab1.admin	DEFT loop - 3 rounds, 3 models wi...	38 minutes ago
scikit-learn	inspector-round2		Draft	lab1.admin	DEFT loop - 3 rounds, 3 models wi...	45 minutes ago
scikit-learn	inspector-round1		Draft	lab1.admin	DEFT loop - 3 rounds, 3 models wi...	an hour ago
scikit-learn	inspector-round3		Draft	lab1.admin	DEFT loop - 3 rounds, 3 models wi...	an hour ago
scikit-learn	inspector-round2		Draft	lab1.admin	DEFT loop - 3 rounds, 3 models wi...	an hour ago
scikit-learn	inspector-round1		Draft	lab1.admin	DEFT loop - 3 rounds, 3 models wi...	an hour ago
scikit-learn	inspector-round3		Draft	lab1.admin	DEFT loop - 3 rounds, 3 models wi...	an hour ago

Tasks: the unit of work

A task is one recorded run of some code: the repository and commit, the container image, the parameters, the console, the metrics and whatever it produced. Two lines turn a script into one. A task can then be cloned, edited and enqueued to a compute resource, where a Klique worker daemon picks it up and runs it on a GPU that is released when it finishes.

```
from clearml import Task

task = Task.init(project_name="Physical AI Inspection", task_name="Train the inspector")
# from here the code, its commit, the parameters, the console and every metric are recorded
```

Pipelines: a fixed sequence of tasks

When the order of work is known in advance, a pipeline chains tasks into a graph. Each step is a task in its own right, the output of one step is passed to the next, each step runs on the compute resource that suits it, and a step whose inputs are unchanged is served from cache. The example code ships this build as a pipeline as well:

```

from clearml import PipelineController

pipe = PipelineController(name="Physical AI Data Factory",
                          project="Physical AI Inspection", version="0.2.0")
pipe.add_parameter("target_per_class", 60)

pipe.add_function_step(name="register", function=register_real,
                      function_return=["hyperdataset_id"],
                      cache_executed_step=True)
pipe.add_function_step(name="generate", function=generate_defects,
                      function_kwargs={"dataset": "${register.hyperdataset_id}"},
                      docker=GPU_IMAGE, execution_queue=GPU_COMPUTE_RESOURCE)
pipe.add_function_step(name="score", function=score_generated, ...)
pipe.add_function_step(name="publish", function=publish_generated, ...)

pipe.start()

```

The screenshot displays the Klique platform interface for managing pipeline runs. On the left, the 'RUNS LIST' shows three runs of the 'Physical AI Data Factory' pipeline, with the first two in 'Running' status and the third 'Completed'. The central area features a visual representation of the pipeline steps: 'Ingest' leads to 'generate', which then branches to 'assemble' and 'inspect'. The right-hand 'RUN INFO' panel provides details for the selected run, including parameters like 'num_images' (8) and 'steps' (30), and sections for 'METRICS', 'ARTIFACTS', and 'MODELS'.

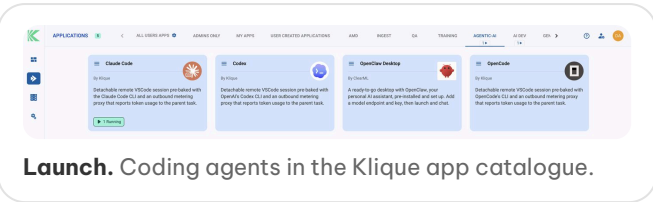
A pipeline run in Klique. Each box is a step that ran as its own task; the run list keeps every execution.

Agent loops: when the next step depends on the last result

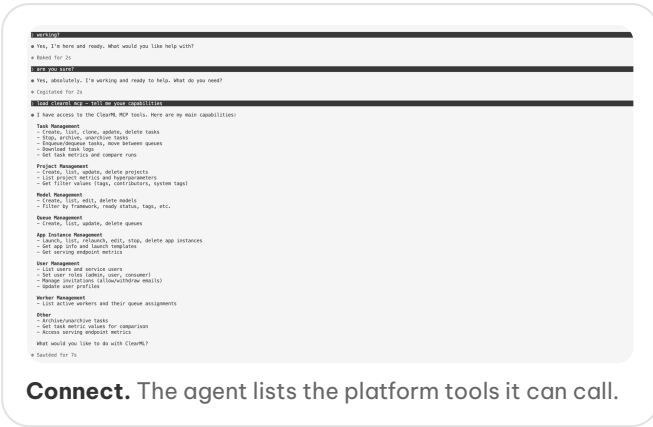
A pipeline repeats the same plan every time. An agent loop decides its next step from what it has just read: how many examples each defect class has, which class scored worst, whether another round is worth its cost. This build uses a loop because those questions change every round. The tasks are the same either way, and so is the record; the difference is who chooses what runs next.

Agents: sandboxed apps with MCP tools

The agent is an app like any other on the platform. A builder launches a coding agent from the app catalogue and it starts in its own secure sandbox: a container session with no GPU and no special access. It reaches the platform through the Klique MCP server, which exposes tasks, projects, models, compute resources and apps as tools the agent can call. Every call runs with the builder's own permissions and quotas, and every task it starts lands in the project like any other.



Launch. Coding agents in the Klique app catalogue.



Connect. The agent lists the platform tools it can call.

It starts with a prompt

Here is the part that surprises most builders: nobody writes a pipeline to run this. The whole loop is started from a coding agent session with a few short prompts, and the agent does the rest. It reads its standing brief from the example repo, writes a task for each stage with that round's settings, enqueues it to a GPU compute resource, reads the result and decides what to do next.

Launching the agent

The agent is the Claude Code app from the catalogue. Its launch form takes a minute, and a few settings matter for a loop like this one:

FIELD	SETTING	WHY
Compute Resource	A CPU compute resource	The agent holds no GPU. Everything heavy is enqueued to GPU compute resources.
Claude Code Session Project	The build's project	The session sits beside every task it launches.
Model	A gateway profile, or the builder's own Anthropic key	By default Claude Code routes through the platform's LLM gateway. An own key goes in the Configuration Vault as ANTHROPIC_API_KEY, with <i>I already have an API key in my Vault</i> ticked.
Skip permission prompts	On	Lets the loop run unattended inside its sandbox. The first prompt still ends with a stop.
Community skills	Off	The repository brings its own skill and standing brief.
Git Access	Empty	The example repository is public. These fields only add credentials for private repositories or pushing.
Advanced: Setup Script	Optional	Can pre-clone the repository. Leaving the clone to the agent keeps the whole run in its transcript.

There is no repository field: the agent clones the example repository itself, as the first line of its first prompt.

Step 1: set up the environment with the agent

Before any GPU work, we work with our agent to set up the environment. The first prompt asks it to find its way around the environment and report back: which platform tools it has through MCP, which compute resources exist and which look like candidates for each role, what the dataset is short of, whether the project already holds tasks, and whether task pods share a storage path. Then it stops, so we can check its report before anything runs.

First, clone <https://github.com/damianerangey-cml/hands-on-labs.git> into `~/hands-on-labs` and read `physical-ai-deft/AGENT.md` completely before running anything. Work from `~/hands-on-labs/physical-ai-deft` from then on.

You are the control plane for an NVIDIA Cosmos AnomalyGen pipeline on this Klique (ClearML) server. You have no GPU and you never will: you drive the platform and enqueue work onto GPU queues.

DRIVE THE PLATFORM THROUGH YOUR MCP TOOLS. This session's Claude Code app configured them. If the MCP server fails to connect, report its name, its URL and the error, and STOP.

LAUNCHING A STAGE: create a draft for that run with `launch.py <stage> --dry-run` and the run's own arguments (`--run-id`, `--gap`, `--num-sdg`, `--baseline`), then enqueue the draft with the MCP and watch it there.

REPORT BACK -- label every claim VERIFIED, INFERRED or UNKNOWN.

- a. Which MCP tools do you have?
 - b. The queues on this server, from the MCP.
 - c. `deft.gap()` -- what the dataset is short of.
 - d. The tasks already in this project, if any.
 - e. Which queue serves each role: `gpu`, `cpu`, `gpu_large` (optional).
 - f. Whether task pods share a storage path for the stages.
- For e and f, show me what you found and ask.

Then STOP. Do not launch a stage yet.

> First, clone <https://github.com/damianerangey-cml/hands-on-labs.git> into `~/hands-on-labs` and read `physical-ai-deft/AGENT.md` completely before running anything. Work from `~/hands-on-labs/physical-ai-deft` from then on.

You are the control plane for an NVIDIA Cosmos AnomalyGen pipeline on this Klique (ClearML) server. You have no GPU and you never will: you drive the platform and enqueue work onto GPU queues.

DRIVE THE PLATFORM THROUGH YOUR MCP TOOLS. This session's Claude Code app configured them. If the MCP server fails to connect, report its name, its URL and the error, and STOP.

LAUNCHING A STAGE: find its seed task with the MCP, CLONE it, change only what the round needs, enqueue the clone. If a stage has no seed task yet, create it once as a draft with `launch.py <stage> --dry-run`.

REPORT BACK -- label every claim VERIFIED, INFERRED or UNKNOWN.

- a. Which MCP tools do you have?
 - b. The queues on this server, from the MCP.
 - c. `deft.gap()` -- what the dataset is short of.
 - d. The seed tasks in this project, if any.
 - e. Which queue serves each role: `gpu`, `cpu`, `gpu_large` (optional).
 - f. Whether task pods share a storage path for the stages.
- For e and f, show me what you found and ask.

Then STOP. Do not launch a stage yet.

The prompt in the session, as first sent (the launch line has since been updated as above). A longer version, with the reasons behind each line, is in the example repo's `PROMPT.md`.

On our environment the agent found its tools, listed every compute resource without guessing which to use, confirmed that the dataset had not been registered and that the project held no tasks yet, then asked before going further:

- Now gap() and the project/seed-task lookups.
Called clearml, ran 1 shell command
- The control plane works, but nothing is set up yet. I stopped before launching anything. I haven't run deft.set_queues() and I haven't created any seed tasks.
~/hands-on-labs already existed. It's a git checkout of the right origin, on main and clean. I read AGENT.md in full (486 lines).

a. MCP tools

- VERIFIED: the mcp_clearml_* server connects; ping returned pong.
- VERIFIED: the deferred tool list includes these for this pipeline:
 - tasks: list_tasks, clone_task, update_task, enqueue_task, create_task, stop_task, download_task_log, get_task_metric_values
 - queues: list_queues, list_workers
 - models and projects: list_models, list_projects
 - apps: launch_app_instance
- UNKNOWN: whether each of those tools works when called. I only called ping, list_queues, list_projects and list_workers.
- VERIFIED: the MCP has no dataset tools, as AGENT.md says.

b. Queues (list_queues, page_size 500, so a short page means this is the full set of 9 visible to me)

- apps_queue is internal and tagged hidden and applications.
- _bootstrap_ is a plain queue.
- Seven queues are tagged profile-connection:

Queue	Display name
1XGPU	Full GPU
0.5XGPU	Half GPU
0.25XGPU	Quarter GPU
0.125XGPU	Eighth GPU
1XCPU	1 CPU Slot
1xGPU-48GB	none

1xGPU-48GB has no display name, so it's the one entry whose name isn't confirmed by a label.

- VERIFIED: all of the above is the API's own output.
- VERIFIED: list_queues doesn't say what GPU or card size sits behind any queue, so I can't tell.

c. deft.gap()

- VERIFIED (run with /usr/local/bin/python3, which imports clearml 2.1.12): counts and gap are empty, with the note "no HyperDataset 'PCB Inspection' on this server — run register first".
- The real data hasn't been registered, so the dataset is short of everything.
- UNKNOWN: whether HyperDatasets are supported at all. The call returned a clean "not found" rather than an endpoint error, but I haven't confirmed support beyond that.
- deft.history() returned no versions. Its models lookup errored with "No projects found when searching for Physical AI Inspection".

d. Seed tasks

- VERIFIED: there are none. The project Physical AI Inspection doesn't exist, and list_projects found nothing matching "Physical", "Inspection" or "deft".
- I haven't created seeds yet. That means running launch.py <stage> --dry-run for each stage, and I'll wait for your go-ahead.

The agent's report. Every claim is labelled verified, inferred or unknown, and it stops to ask.

Every run will look different. The prompt can be adapted to each environment, and the agent's report depends on what it finds there: different compute resources, GPUs, storage and starting data. That is why it asks rather than assumes.

Step 2: answer for the environment, then hand over the goal

The agent asks for what it cannot find out for itself: which compute resource should serve each role, and whether task pods share a storage path. Our answers are specific to our environment, and answers on another environment will differ. We sent them in the same message as the goal, with a condition: run a quick check of what a task receives (a GPU, the shared storage path, the NVIDIA and Hugging Face credentials), and only carry on if that check passes.

Answers:

- e. deft.set_queues(gpu="1XGPU", cpu="1XCPU", gpu_large="1xGPU-48GB")
- f. Task pods on the GPU queues share a model cache at /cache.

Go ahead: record e and run probe_parity.py once. Report what the probe found.

If the probe shows a GPU, /dev/shm, the shared path and both credentials, carry on without waiting for me:

Get every defect class to 60 examples using NVIDIA's Cosmos AnomalyGen.
Budget: 3 rounds, and stop earlier if a round adds nothing.
Train the control on real images only before you generate anything.
Tell me what you decided and why, including the calls that did not pay off. And do not let the dataset claim more than it has.

If anything fails, report it and stop.

```

f. Shared storage across task pods
- UNKNOWN. I haven't run a task on this server. The parity probe is the thing that would show it, and it launches a throwaway GPU task, so I haven't run it.
- DEFT_CACHE defaults to /cache in this repo. That's the repo's own convention and says nothing about this server.
- Question: is there a path mounted on every task pod, such as a PVC or a bind mount? If so, where? If not, only single-task launch.py rounds runs will work.

Also unconfirmed: AGENT.md says NGC_KEY, NGC_API_KEY and HF_TOKEN must reach the task's environment through the vault or a secret. I haven't checked that, so it's UNKNOWN. Tell me if you want the probe run to find out.

Once you've answered e and f, the next steps are to record the queues with deft.set_queues(), create the seed tasks as drafts, and run the parity probe if you want it.

* Brewed for 33s · done 11:05 AM

* recap: I'm setting up the Cosmos AnomalyGen pipeline on this server. The platform tools work, but there are no seed tasks and no dataset yet, and I've stopped before launching anything. Next, I need you to say which queue serves each role (gpu, cpu, gpu_large) and whether task pods share a storage path. (disable recaps in /config)

> Answers:
  e. deft.set_queues(gpu="1xGPU", cpu="1xCPU", gpu_large="1xGPU-48GB")
  f. Task pods on the GPU queues share a model cache at /cache.

Go ahead: record e, create the seed tasks with launch.py <stage> --dry-run, and run probe_parity.py once. Report what the probe found.

If the probe shows a GPU, /dev/shm, the shared path and both credentials, carry on without waiting for me:

Get every defect class to 60 examples using NVIDIA's Cosmos AnomalyGen. Budget: 3 rounds, and stop earlier if a round adds nothing. Train the control on real images only before you generate anything. Tell me what you decided and why, including the calls that did not pay off. And do not let the dataset claim more than it has.

If anything fails, report it and stop.

● Running cd ~/hands-on-labs/physical-ai-deft && /usr/local_ . 3s
  L $ cd ~/hands-on-labs/physical-ai-deft && for s in register generate evaluate improve publish train finetune; do echo "== $s"; /usr/local/bin/python3 launch.py $s --dry-run 2>&1 | tail -6; done (4s · 9 lines)
    (ctrl+b to run in background)

```

The same message in the session. The agent starts by creating a draft task for each stage.

Credentials are set up ahead of time. The NVIDIA and Hugging Face keys the stages need were set up ahead of time by an administrator, in the Configuration Vault or a platform secret, so they reach every task without appearing in a prompt, a task parameter or the agent's transcript. The agent only checks that they arrived.

From there the builder can talk to the loop while it runs. A fair follow-up, mid-run:

```

excess_solder keeps scoring worst. Work out whether that is fixable,
and if it is not, stop spending GPU on it.

```

The prompts live in the [example repo](#) alongside the agent's standing brief, [AGENT.md](#), which tells it plainly that the stages are operations to choose between, not a script to follow: the agent picks the calls, their inputs and the moment to stop.

NVIDIA and Klique, side by side

Everything on the left is NVIDIA's, used as shipped. Everything on the right is a standard Klique capability; we built none of it for this guide.

LAYER	NVIDIA	KLIQUE
Generate	Cosmos AnomalyGen: inpaints a defect onto a clean board, steered by a mask	A GPU compute resource and a tracked task per run
Place	Automatic Mask Placement: puts the mask where the board's CAD says a fault can occur	Masks saved on the task as debug samples
Evaluate	nn_score against real examples; Cosmos Reason as a plausibility check	Cosmos Reason launched as a NIM endpoint from the app catalogue, with idle-stop
Data	Sample images, masks and clean references	HyperDatasets: searchable labels per image and a new version every round
Train	A few-shot adapter for the generator; any inspection model	A larger-memory compute resource for the fine-tune; the model registry for results
Drive	The DEFT loop: evaluate, generate, mine, retrain, gate	A coding agent from the app catalogue, working through the MCP server

One round, stage by stage

Let's walk through one round of the loop. Nobody runs these by hand. Once the builder has given the goal and the budget, the agent runs every stage itself, round after round, and every stage follows the same rhythm: **launch** the task, **wait** for it while it runs on a GPU, **read** what it produced, **decide** what comes next. That rhythm repeats until the target is met or the budget is spent. Each stage below covers the work, the reason for it, how Klique runs it, and the agent's part. The builder does not write these snippets: they show what runs underneath each stage, issued by the agent through its MCP tools and the helpers in the [example code on GitHub](#).

01 Register the real data

What. The real inspection photos, their defect masks and a few clean reference boards are loaded once into a HyperDataset, Klique's dataset type for images and their labels.

Why. Every later stage asks questions of this data, such as how many solder bridges it holds. A folder of files cannot answer that without someone opening every image.

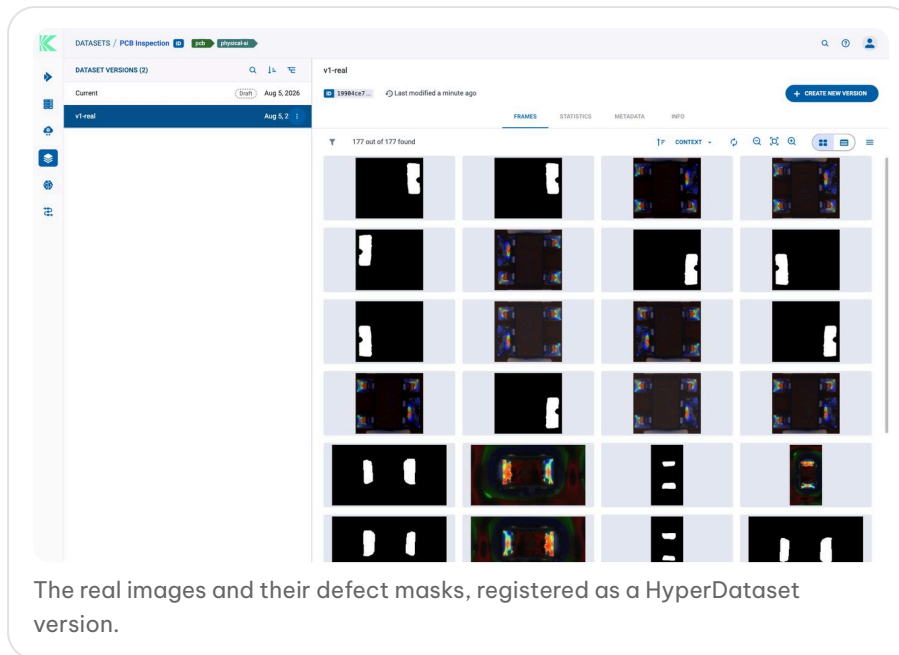
How. Each image becomes a frame carrying its defect label and metadata, and the set is published as version one. From then on Klique answers questions about the data directly, with no download.

Agent. Checks whether a published version already exists. This stage runs once; later rounds skip it.

```
from clearml import Task
import hyperdataset as hd          # helper module in the example repo

Task.init(project_name="Physical AI Inspection",
          task_name="Register the real data",
          task_type=Task.TaskTypes.data_processing)

dataset = hd.get_or_create_dataset("PCB Inspection")
version = hd.create_draft(dataset, "v1-real")
hd.add_frames(version, [hd.make_frame(uri, labels=[defect], meta=meta)
                       for uri, defect, meta in real_images])
hd.commit(version, publish=True)
```



KLIQUE: HYPERDATASETS

02 Count the examples per defect

What. Count the trusted examples of each defect class and compare each count with the target.

Why. The difference decides what to generate. GPU time goes to the classes that are short, and none is spent on a class already at target.

How. One query against the dataset's labels, answered by Klique without opening an image. It is cheap enough to ask on every round, and it works on data that is not allowed to leave its storage.

Agent. Runs the count at the start of every round, works out the shortfall per class, and decides whether a round is worth its GPU time.

```
latest = hd.latest_published(dataset)
counts = hd.stats(latest["id"])["labels"]      # per-label counts, computed by Klique

shortfall = {label: TARGET - n for label, n in counts.items() if n < TARGET}
```

KLIQUE: HYPERDATASET QUERIES

03 Generate what is missing

What. NVIDIA Cosmos AnomalyGen paints the missing defects onto clean boards.

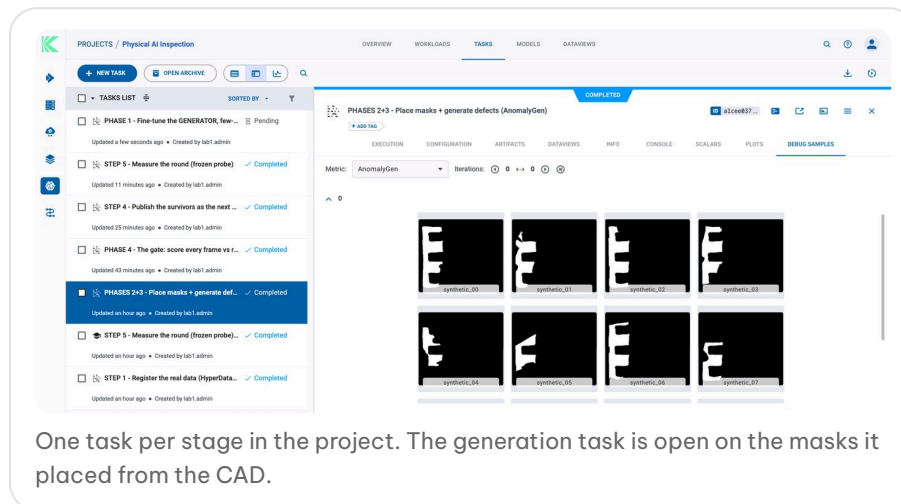
Why. A generated defect is only useful if it could happen on the real board. Each defect is painted into a mask placed where the board's CAD says that fault can occur, so a solder bridge only lands on real pads.

How. NVIDIA's container runs unmodified as a Klique task. Klique supplies the GPU, the cached model weights and the credentials, and releases the GPU when the task ends.

Agent. Creates a draft generation task carrying this round's request (how many of each defect, from the shortfall), enqueues it to a GPU compute resource, then waits, checking the task's status through MCP until generation completes.

```
task = Task.create(
    project_name="Physical AI Inspection",
    task_name="Place masks and generate defects",
    repo=REPO, branch="main",
    script="physical-ai-deft/anomalygen_generate.py",
    docker="nvcr.io/nvidia/paidf-anomalygen:1.0.1") # NVIDIA's container, as shipped

Task.enqueue(task, queue_name=GPU_COMPUTE_RESOURCE)
```



KLIQUE: GPU COMPUTE RESOURCES, TRACKED TASKS, A SHARED MODEL CACHE

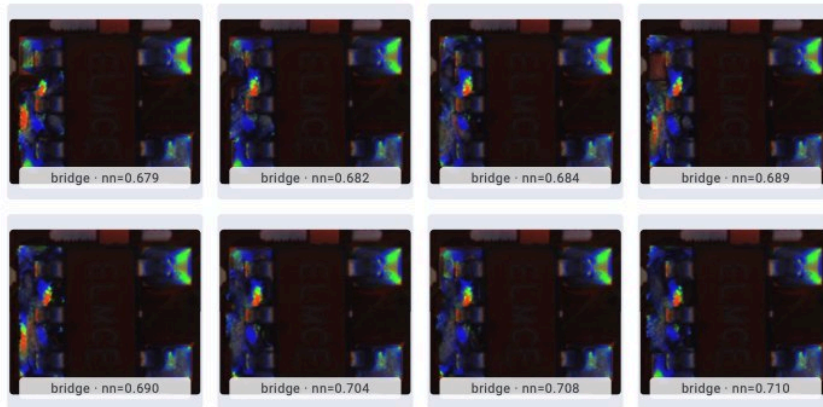
04 Gate every frame

What. Score every generated frame against the real examples of its class.

Why. A generator will sometimes return a plausible picture that does not contain the defect asked for. Training on those frames would teach the inspector the wrong lesson.

How. A frame that clears the threshold earns its label. One that does not is held for review and counts toward no target, so a sizeable share of each round is expected to be held back. Cosmos Reason can be launched as an endpoint for a second opinion on plausibility, and stops itself when idle.

Agent. Enqueues the scoring task, waits for it to complete, then reads the scores per class. If one class keeps scoring poorly, it decides whether to try different settings or to stop spending GPU time on it.



Accepted frames on the gate task, each with its score against real data.

KLIQUE: TASKS, DEBUG SAMPLES, NIM ENDPOINTS

05 Publish a new version

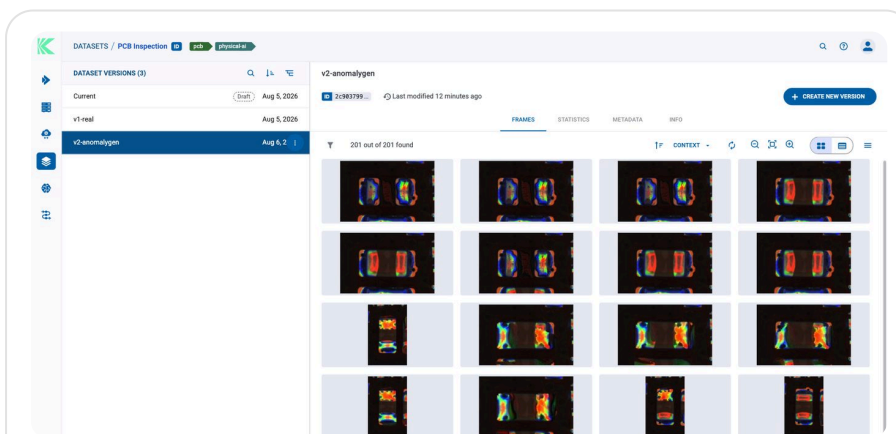
What. The frames that passed become the next version of the dataset.

Why. Training always reads a fixed version, so a result can be reproduced and one round compared with the next.

How. The new version inherits every frame from its parent and adds the accepted ones. Each generated frame keeps a link to the clean board, the mask and the settings that produced it.

Agent. Enqueues the publish of the accepted frames only, waits for the new version to land, then confirms the new counts.

```
parent = hd.latest_published(dataset)
version = hd.create_draft(dataset, hd.next_version_name(dataset),
                          parent=parent["id"]) # inherits the parent's frames
hd.add_frames(version, accepted_frames)        # each carries board, mask, settings
hd.commit(version, publish=True)
```



The next dataset version, with the generated defects that passed the gate.

KLIQUE: DATASET VERSIONS

06 Retrain and measure

What. Retrain the inspection model on the enriched dataset and measure it.

Why. A better inspector is the point of the whole loop. A baseline trained on real images only sits beside it, which shows whether the generated data helped.

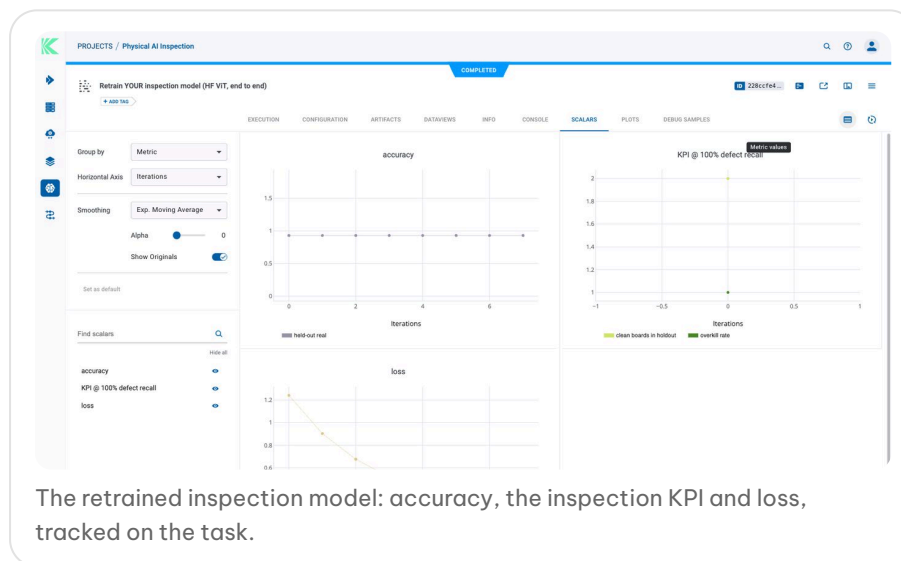
How. Training runs as a task on a GPU, with accuracy and loss plotted as it goes. The finished model is stored in the model registry, linked to the dataset version it learned from.

Agent. Enqueues the training task, then waits: it checks the task's status through its MCP tools until training completes, holding no GPU while it waits. It then reads the metrics, compares the result with the baseline, reports what changed, and decides whether to go round again.

```
from clearml import Task, OutputModel

task = Task.init(project_name="Physical AI Inspection",
                 task_name="Train the inspector")
logger = task.get_logger()
logger.report_scalar("accuracy", "held-out", value=accuracy, iteration=0)

model = OutputModel(task=task, name="inspector", framework="scikit-learn")
model.update_weights(weights_filename=model_path)
```



KLIQUE: MODEL REGISTRY, EXPERIMENT TRACKING

Then the agent goes round again

One pass through the stages is one round, and the loop iterates. Back in its sandbox, the agent counts the examples once more and makes the call. If a class is still short and the budget allows, it starts the next round from stage 02, adjusting its requests in the light of the last round's scores. If the target is met, the budget is spent or a round added little, it stops and reports its decisions and the reasons for them. The builder reads the summary; the agent did the driving.

```
Working!
• Yes, I'm here and ready. What would you like help with?
• Bailed for 7s

Are you sure?
• Yes, absolutely. I'm working and ready to help. What do you need?
• Copitated for 2s

Load clearml.mcp - tell me your capabilities
• I have access to the ClearML MCP tools. Here are my main capabilities:

Task Management
- Create, list, clone, update, delete tasks
- Stop, archive, unarchive tasks
- Engender/queue tasks, move between queues
- Download task logs
- Get task metrics and compare runs

Project Management
- Create, list, update, delete projects
- List project metrics and hyperparameters
- Get filter values (tags, contributors, system tags)

Model Management
- Create, list, edit, delete models
- Filter by framework, ready status, tags, etc.

Queue Management
- Create, list, update, delete queues

App Instance Management
- Launch, list, relaunch, edit, stop, delete app instances
- Get app info and launch templates
- Get serving endpoint metrics

User Management
- List users and service users
- Set user roles (admin, user, consumer)
- Manage invitations (allow/withdraw emails)
- Update user profiles

Worker Management
- List active workers and their queue assignments

Other
- Archive/unarchive tasks
- Get task metric values for comparison
- Access serving endpoint metrics

What would you like to do with ClearML?
• Sautéed for 7s
```

The agent's sandbox. The same session drives every round through its platform tools, holding no GPU of its own.

TASKS LIST		SORTED BY	
☐	 Train the inspector	✓ Completed	
Updated 4 minutes ago • Created by lab1.admin			
☐	 Publish the survivors as the next HyperDat...	✓ Completed	
Updated 12 minutes ago • Created by lab1.admin			
☐	 AnomalyGen -- search, keep best, filter an...	✓ Completed	
Updated 40 minutes ago • Created by lab1.admin 2 iterations			
☐	 AnomalyGen -- score against the real data	✓ Completed	
Updated an hour ago • Created by lab1.admin			
☐	 AnomalyGen -- place masks and generate	✓ Completed	
Updated an hour ago • Created by lab1.admin			
☐	 Train the inspector	✓ Completed	
Updated 2 hours ago • Created by lab1.admin			
☐	 Register the real data (HyperDataset v1-re...	✓ Completed	
Updated 2 hours ago • Created by lab1.admin			

Its work, round by round. Tasks the agent launched, in the project's task list.

Design patterns worth reusing

These are the patterns we would carry into any agent-driven loop on the platform, not only this one.

Keep the loop's memory in the data

The agent reads its state from the dataset's label counts, not from a file on one machine. An agent that restarts asks what is true now and carries on.

Generated data earns its label

A measured distance to real data decides what counts, not the generator's intent. Otherwise the loop reads its own optimism back as fact.

The agent holds no GPU

It enqueues work through the same compute resources, quotas and access as its user, so it cannot exceed that user's permissions.

Ask, do not guess

Names that must match the estate, such as which compute resource has a large-memory GPU, are confirmed with a person once and stored.

Let the launcher write the task

Each run's task is written by the repository's launcher as a draft, carrying the container image, the settings that make a stage run, and that run's own arguments. The agent enqueues it and follows it through to the end; it never hand-builds a task.

Train a baseline first

A baseline on real data only is what makes "the synthetic data helped" a claim that can be checked.

Bound the spend before starting

A round budget and a stop rule are part of the goal. Endpoints the loop starts carry an idle timer, so a forgotten one releases its GPU.

Report what was measured

Asked versus delivered, accepted versus pending, per class. A flat result on a small holdout is reported as such.

What it takes to run

ITEM	REQUIREMENT
Generation and scoring	One mid-range GPU per task; a batch of images takes minutes
Generator fine-tune	One large-memory GPU, used only for that stage; NVIDIA's published adapter lets the loop start before it finishes
The agent	CPU only: a coding agent session from the app catalogue
Model weights	NVIDIA's checkpoints, cached once on a shared volume
Starting data	A small set of real images per defect class, their masks, a few clean boards and the board's CAD
Platform features	HyperDatasets, compute resources, tasks, the model registry, the app catalogue and the MCP server

Results and next steps

Here is what we found: the loop ran end to end with an agent at the controls, and delivered each piece it set out to build.

- **Convincing defects from a small sample.** Starting from a small set of real images, Cosmos AnomalyGen produced solder bridges, excess solder and missing components on real boards, each placed where the board's CAD allows that fault.
- **A dataset that grew only with frames that earned their place.** The gate kept the frames that scored well against real examples and held the rest for review, so each new version can be trusted for training.
- **A retrained inspection model.** The inspector was retrained on the enriched dataset and stored in the model registry beside its baseline, ready to compare.
- **An agent that ran it unattended.** From one plain-English goal, the agent chose the requests, launched the GPU work, judged the results and stopped within its budget.

Taking it to production

This run used NVIDIA's public sample set, which is sized to prove the method. The same loop scales with the data it is given:

- **More real data.** A production line's own images, and a larger held-out set, turn the inspection KPI into a number to steer by.
- **More rounds.** Some defect classes, excess solder among them, benefit from extra rounds and more real examples to learn from.
- **More for the agent to tune.** NVIDIA's method has a place for an agent to choose generation settings per frame, a natural next step for this build.

Adapt it

T0	CHANGE
Use different boards or parts	Register that data and its CAD; start with NVIDIA's published adapter, then fine-tune one
Inspect something that has no CAD	Describe the defect location in text; a vision model and a segmenter place the mask
Train a different inspector	Swap the trainer image, for example NVIDIA TAO; the compute resource and the workflow stay the same
Run it unattended	Give the agent a round budget and a stop rule; review the task list and the dataset versions the next morning
Drive it without an agent	Run the same stages from a script or a pipeline; the stages are identical

Get started

Everything in this guide is available to try: a hosted lab with the environment prepared, the example code on GitHub, and NVIDIA's open-source components.

Try it on Klique

The Klique hands-on labs provide a ready environment with the GPUs, the data and the agent in place, taking a builder from sign-in to a retrained model. Or clone the example code and run it on any Klique environment.

[Launch a hands-on lab](#)
[Get the code on GitHub](#)

RESOURCE	WHAT IT IS	LINK
Klique hands-on labs	A hosted Klique environment, ready for this build	clearml-hol.com
Example code	The stage scripts and the launcher used for this build	damianerangey-cml/hands-on-labs
NVIDIA Cosmos	NVIDIA's world foundation models for physical AI	nvidia.com/ai/cosmos/docs
Blueprint announcement	NVIDIA's introduction to the Physical AI Data Factory	nvidianews.nvidia.com
NVIDIA Physical AI Data Factory	NVIDIA's blueprint: workflows and cookbooks	NVIDIA/physical-ai-data-factory
Cosmos AnomalyGen	The defect generator and Automatic Mask Placement, open source	NVIDIA/paidf-anomalygen
Cosmos Predict	The world foundation model underneath	nvidia-cosmos/cosmos-predict2
NVIDIA agent skill	NVIDIA's own agent skill for this inspection loop	NVIDIA/skills
ClearML SDK	The open-source SDK behind tasks, datasets and models	clearml/clearml

Klique AI · Builders series · NVIDIA Cosmos on Klique. Images come from NVIDIA's sample PCB inspection set; results depend on the data used. NVIDIA, Cosmos and NIM are trademarks of NVIDIA Corporation.