AgileTaskHeron

Robot Orchestration for MES

Deployment & Integration Manual — CPU & GPU Editions

CPU edition: version 1.0.2 | GPU edition (RAPIDS): version 1.0.2

Container products for AWS Marketplace

Contents

1. Overview

AgileTaskHeron is a task-orchestration analysis service for manufacturing execution systems (MES) and robotic work-cell environments. Given a graph of tasks, skills, people, or robots and their relationships, the service computes priority rankings, work-group partitions, and robot-to-task eligibility using graph analysis algorithms (PageRank, Louvain community detection).

This product is delivered as two separate container images — a CPU edition and a GPU edition (RAPIDS-accelerated) — sold under the same AWS Marketplace listing as two ECR repositories. Both editions expose an identical HTTP API and return identical response schemas, so client applications can switch between them, or mix them across sites, without code changes.

2. Editions at a Glance (CPU vs. GPU)

	CPU Edition	GPU Edition
Backend libraries	`pandas, numpy, networkx`	`cudf, cugraph (RAPIDS)`
GPU required	No	Yes (NVIDIA, CUDA 12)
Base image	`python:3.12-slim`	`rapidsai/base:26.08-cuda12-py3.11`
Cold-start warmup	Not applicable	Yes — primes CUDA context/RMM at startup
"backend" field (GET /)	`"cpu"`	`"gpu"`
Best fit	Small/medium graphs, cost-sensitive lines	Large graphs, high-throughput sites

Both editions are licensed under the same contract dimensions (see Section 10) and may be deployed side by side — for example, the GPU edition at a high-volume factory site and the CPU edition at smaller lines within the same contract.

3. Architecture & Delivery Model

Both editions are delivered as OCI container images distributed through AWS Marketplace-managed Amazon ECR repositories — one repository per edition (`agiletaskheron-cpu` and `agiletaskheron-gpu`). Each image is built with a multi-stage Docker build:

- **Builder stage** — compiles the application source into a native Python extension module using Nuitka. The C/C++ toolchain and the original .py source exist only in this throw-away build stage.
- **Runtime stage** — contains only the compiled extension module, its runtime dependencies, and uvicorn. No application source code ships inside the final image.

CPU edition

Both stages build on `python:3.12-slim`. The final image runs as a non-root user (uid 47099074) and requires no GPU.

GPU edition

Both stages build on the same RAPIDS base image (`rapidsai/base:26.08-cuda12-py3.11`), since cudf/cugraph and their CUDA dependencies exist only inside that base — a second conda environment is intentionally not created, to keep the image from duplicating the RAPIDS stack. Only the FastAPI/uvicorn service layer is added on top of the base environment. The compiled module's ABI is tied to the base image's Python minor version (currently 3.11) and must match the CUDA driver on the host (CUDA 12).

At container startup, the GPU edition runs a one-time warmup routine that builds a small cudf.DataFrame and exercises the cugraph PageRank/Louvain kernels. This initializes the CUDA context and the cudf/RMM memory pool ahead of the first real request, avoiding a slow first call. Set the environment variable `ATH_WARMUP=0` to skip this (for example, when debugging on a host without a GPU).

> *The GPU container is intended to run as the `rapids` user built into the RAPIDS base image (see run.sh), matching the permissions the RAPIDS environment expects.*

Runtime stack — CPU edition

Component	Version
Python	`3.12`
FastAPI	`0.121.2`
uvicorn	`0.38.0`
pandas	`3.0.5`
numpy	`2.5.2`
networkx	`3.6.1`
scipy	`1.17.1`

Runtime stack — GPU edition

Component	Version
Base image	`rapidsai/base:26.08-cuda12-py3.11`
Python	`3.11`
CUDA	`12.x`
FastAPI	`0.121.2`
uvicorn	`0.38.0`
cudf	`26.8.0`

cugraph	`26.8.0`
cupy	`14.2.0`
numba / numba-cuda	`0.64.0 / 0.30.4`
pandas	`3.0.5`
numpy	`2.4.6`

The GPU image also includes the wider RAPIDS/Dask ecosystem present in the base image (dask-cudf, cuml, xgboost, and related packages) even though the application code only imports cudf and cugraph directly. This is a byproduct of building on the shared RAPIDS base rather than a per-package install, and explains why the GPU image is substantially larger than the CPU image.

4. System Requirements

Resource	CPU Edition	GPU Edition
vCPU	1 min / 2 recommended	2 min / 4 recommended
Memory	1 GiB min / 2 GiB recommended	8 GiB min / 16 GiB recommended
GPU	Not required	1x NVIDIA GPU, CUDA 12 driver
Architecture	linux/amd64	linux/amd64
Orchestrator	Amazon ECS or EKS	Amazon ECS or EKS with GPU-enabled instances (e.g. g4dn/g5)

Sizing above reflects the baseline analysis workload. Very large graphs (tens of thousands of nodes/edges) may require additional memory; benchmark against your own dataset before production sizing. GPU memory headroom should account for the RAPIDS memory pool reserved during the startup warmup routine.

5. Quick Start — CPU Edition

Build the image locally

```
docker build -t agiletaskheron-cpu:local .
```

Run the container

```
docker run --rm -p 8080:8080 agiletaskheron-cpu:local
```

Verify it is running

```
curl http://localhost:8080/health
# {"status": "healthy"}

curl http://localhost:8080/examples/human
curl http://localhost:8080/examples/robot
```

No environment variables or external configuration files are required to start the service. All example endpoints work out of the box using built-in sample data.

6. Quick Start — GPU Edition

> *Requires a host with an NVIDIA GPU, the NVIDIA Container Toolkit, and a CUDA 12 compatible driver installed.*

Build the image locally

```
docker build -t agiletaskheron-gpu:local .
```

Run the container

```
docker run --rm --gpus all -p 8080:8080 --user rapids agiletaskheron-gpu:local
```

The `--gpus all` flag is required — without it, cudf/cugraph fail to initialize a CUDA context and the container will error out during the startup warmup. The `--user rapids` flag runs the container as the RAPIDS base image's built-in user account.

Startup warmup

On startup, the container logs a warmup line once the CUDA context, cudf/RMM memory pool, and cugraph kernels are primed:

```
[warmup] initializing cudf/cugraph/CUDA ...
[warmup] GPU stack ready in 4.82s
```

Set `ATH_WARMUP=0` as a container environment variable to skip this step (e.g. for CPU-only debugging of the GPU image without a GPU attached). The container will still start, but the first real request will pay the warmup cost instead.

Verify it is running

```
curl http://localhost:8080/health
# {"status": "healthy"}

curl http://localhost:8080/
# {"service": "AgileTaskHeron", "backend": "gpu", "status": "ok"}

curl http://localhost:8080/examples/human
curl http://localhost:8080/examples/robot
```

7. API Reference (shared by both editions)

All endpoints accept and return JSON. There is no authentication layer built into the container itself — access control should be enforced at the network / API gateway layer (e.g. an ALB, API Gateway, or the orchestrator's security group rules). Both editions expose the exact same routes and response schemas; the only functional difference is the value of the `backend` field returned by `GET /`.

GET `/`

Service identity check. Returns the backend type currently running (useful for confirming CPU vs. GPU edition in mixed deployments).

Response (excerpt)

```
CPU edition:
{"service": "AgileTaskHeron", "backend": "cpu", "status": "ok"}

GPU edition:
{"service": "AgileTaskHeron", "backend": "gpu", "status": "ok"}
```

GET `/health`

Liveness/readiness probe. Configure this as the container health check in your ECS task definition or Kubernetes probe.

Response (excerpt)

```
{"status": "healthy"}
```

POST `/analyze/human`

Analyzes a task–skill–person graph. Returns PageRank-based priority scores and Louvain-partitioned work-group bundles (task, required skills, available people).

Request body

```
{
  "edges": [
    {"src": "Person1", "dst": "Task1", "weight": 1.0},
    {"src": "Task1", "dst": "Skill1", "weight": 0.6}
  ]
}
```

Response (excerpt)

```
{
  "priority_bundles": [
    {"priority": 1, "task": "Task1", "score": 0.42,
     "required_skills": ["Skill1"], "available_people": ["Person1"], ...}
  ],
  "pagerank_scores": [...],
  "graph_stats": {"nodes": 4, "edges": 3}
}
```

POST `/analyze/robot`

Analyzes a robot–task eligibility graph. Matches robots to tasks based on required skills, minimum software/hardware version, hardware condition, and battery level, then ranks tasks by dependency-aware priority.

Request body

```
{
  "edges": [
    {"src": "Robot001", "dst": "Task001", "edge_type": "JOB_COMPLETION_RATING", "weight": 0.92},
    {"src": "Task001", "dst": "Skill001", "edge_type": "REQUIRES_SKILL", "weight": 0.8}
  ],
  "max_robots_per_task": 10
}
```

Response (excerpt)

```
{
  "priority_bundles": [
    {"priority": 1, "task": "Task001", "priority_score": 0.5,
     "eligible_robot_count": 2, "eligible_robots": [...], ...}
  ],
  "graph_stats": {"nodes": 8, "edges": 12, "robots": 3, "tasks": 2}
}
```

GET `/examples/human`

Runs /analyze/human against a built-in sample dataset. Useful for smoke-testing a fresh deployment without preparing your own graph data.

GET `/examples/robot`

Runs /analyze/robot against a built-in sample dataset. Accepts an optional ?max_robots_per_task= query parameter.

8. Deploying on AWS (ECS / EKS)

Container port

The container listens on **port 8080**. Map this port in your ECS task definition or Kubernetes Service/Ingress.

Example ECS task definition (excerpt)

```
{
  "containerDefinitions": [
    {
      "name": "agiletaskheron-cpu",
      "image": "<marketplace-ecr-uri>:<tag>",
      "portMappings": [{"containerPort": 8080, "protocol": "tcp"}],
      "cpu": 1024,
      "memory": 2048,
      "healthCheck": {
        "command": ["CMD-SHELL", "curl -f http://localhost:8080/health || exit 1"],
        "interval": 30,
        "timeout": 5,
        "retries": 3
      }
    }
  ]
}
```

Example Kubernetes probe (excerpt)

```
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 15
readinessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 10
```

No environment variables are required for basic operation. The container writes nothing to disk and does not require a persistent volume.

The CPU container runs as a non-root user by default; no additional securityContext changes are required to satisfy non-root pod security policies.

GPU edition — additional configuration

GPU tasks/pods must request a GPU resource and run on GPU-backed instances (e.g. g4dn, g5 instance families). Because of the startup warmup routine, allow a longer initial health-check grace period than the CPU edition.

Example ECS task definition (excerpt, GPU)

```
{
  "containerDefinitions": [
    {
      "name": "agiletaskheron-gpu",
      "image": "<marketplace-ecr-uri>:<tag>",
      "portMappings": [{"containerPort": 8080, "protocol": "tcp"}],
      "resourceRequirements": [{"type": "GPU", "value": "1"}],
      "cpu": 4096,
      "memory": 16384,
      "healthCheck": {
        "command": ["CMD-SHELL", "curl -f http://localhost:8080/health || exit 1"],
        "interval": 30,
        "timeout": 5,
        "retries": 3,
        "startPeriod": 60
      }
    }
  ]
}
```

Example Kubernetes pod spec (excerpt, GPU)

```
resources:
  limits:
    nvidia.com/gpu: 1
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 45
  periodSeconds: 15
readinessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 45
  periodSeconds: 10
```

The GPU node group (EKS) or capacity provider (ECS) must have the NVIDIA device plugin / GPU driver support enabled. Refer to the AWS documentation for your chosen GPU-enabled AMI or Bottlerocket-with-GPU node group.

9. Health Checks & Monitoring

Use `GET /health` as the container-level liveness check. It returns HTTP 200 with `{"status": "healthy"}` as long as the FastAPI process is responsive; it does not perform any external dependency checks, since the service has none.

For deeper smoke-testing after a deployment or version update, call `GET /examples/human` and `GET /examples/robot` and confirm the response contains a non-empty `priority_bundles` array.

10. Licensing & Contract Dimensions

This product is offered under AWS Marketplace contract-based pricing. Licensing is metered per the following contract dimensions:

API identifier	Display name	Definition
`production_line_license`	Production Line License	Annual license per production line.
`factory_site_license`	Factory Site License	Annual license for unlimited production lines at one factory site.

The CPU and GPU editions are licensed under the same contract dimensions and can be mixed within a single contract (e.g. CPU edition for smaller lines, GPU edition for high-throughput sites).

11. Troubleshooting

Container exits immediately on start

Confirm the image was built for linux/amd64. On Apple Silicon, rebuild with `docker buildx build --platform linux/amd64`.

/health returns connection refused

Confirm port 8080 is mapped/exposed and that no host firewall or security group rule is blocking inbound traffic on that port.

422 error on /analyze/* endpoints

The edges array was empty or malformed. Each edge requires src, dst, and weight (and edge_type for /analyze/robot).

400 error on /analyze/robot

One or more required edge columns (src, dst, edge_type, weight) were missing, or a non-numeric weight value was supplied.

GPU edition fails during startup warmup / CUDA initialization error

Confirm the container was started with `--gpus all` (Docker) or a GPU resource request (ECS/Kubernetes), and that the host driver supports CUDA 12. Set `ATH_WARMUP=0` only for temporary CPU-only debugging of the GPU image — it will not perform GPU analysis correctly without a GPU.

GPU edition: 'no CUDA-capable device is detected'

The container did not receive GPU access from the orchestrator. Verify the ECS capacity provider or Kubernetes node has the NVIDIA device plugin installed and that the task/pod spec requests a GPU resource.

GPU edition image pull is very slow

The RAPIDS base image is large (tens of GB). Ensure the pulling instance is in the same AWS Region as the Marketplace-managed ECR repository, and consider pre-warming the image on frequently recycled

instances/AMIs.

12. Support

For technical support, integration questions, or guidance choosing between the CPU and GPU editions for your deployment, contact the seller through the support channel listed on the AWS Marketplace product page.