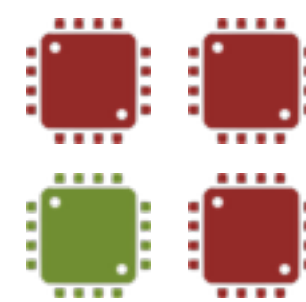
Approximate Computation with Topaz

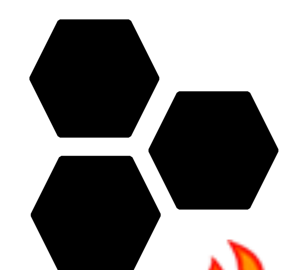
Sara Achour and Martin Rinard
MIT EECS & CSAIL

The world contains a lot of approximate hardware

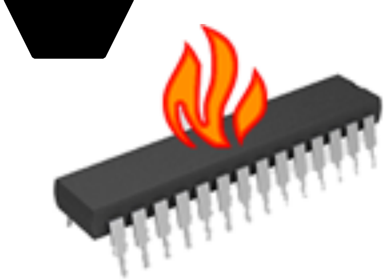
- <u>**What is approximate hardware?**</u>
  - Hardware that crashes often
  - Sometimes produces wildly inaccurate results,
  - Often produces slightly inaccurate results

- <u>**Approximate hardware exists today**</u>
  - Manufacturing defects
  - Old machines
  - Aggressive operating conditions
  - Relaxed hardware mechanisms
  - Emerging hardware technologies





- <u>**Approximation is a hardware design point**</u>
  - **Energy** and **performance** savings



Why is using approximate hardware hard?

- **Prior techniques require error model**
  - Assumptions on fault characteristics
  - Narrow applicability
- **In the wild, hardware faults can be complex**
  - Correlated with other faults
  - Dependent on hardware state
  - Yield small errors, large errors or crashes
- **Fault characteristics of future hardware unknown**
- **Need system that generalizes broadly across many approximate computing platforms.**

Topaz

- **Topaz** is a enables the deployment of programs on approximate hardware ***with no hardware specification***
    - Operates on **hardware** with ***complex*** fault characteristics
    - Targets **programs** that are robust and tolerant of error

Topaz: Key Features

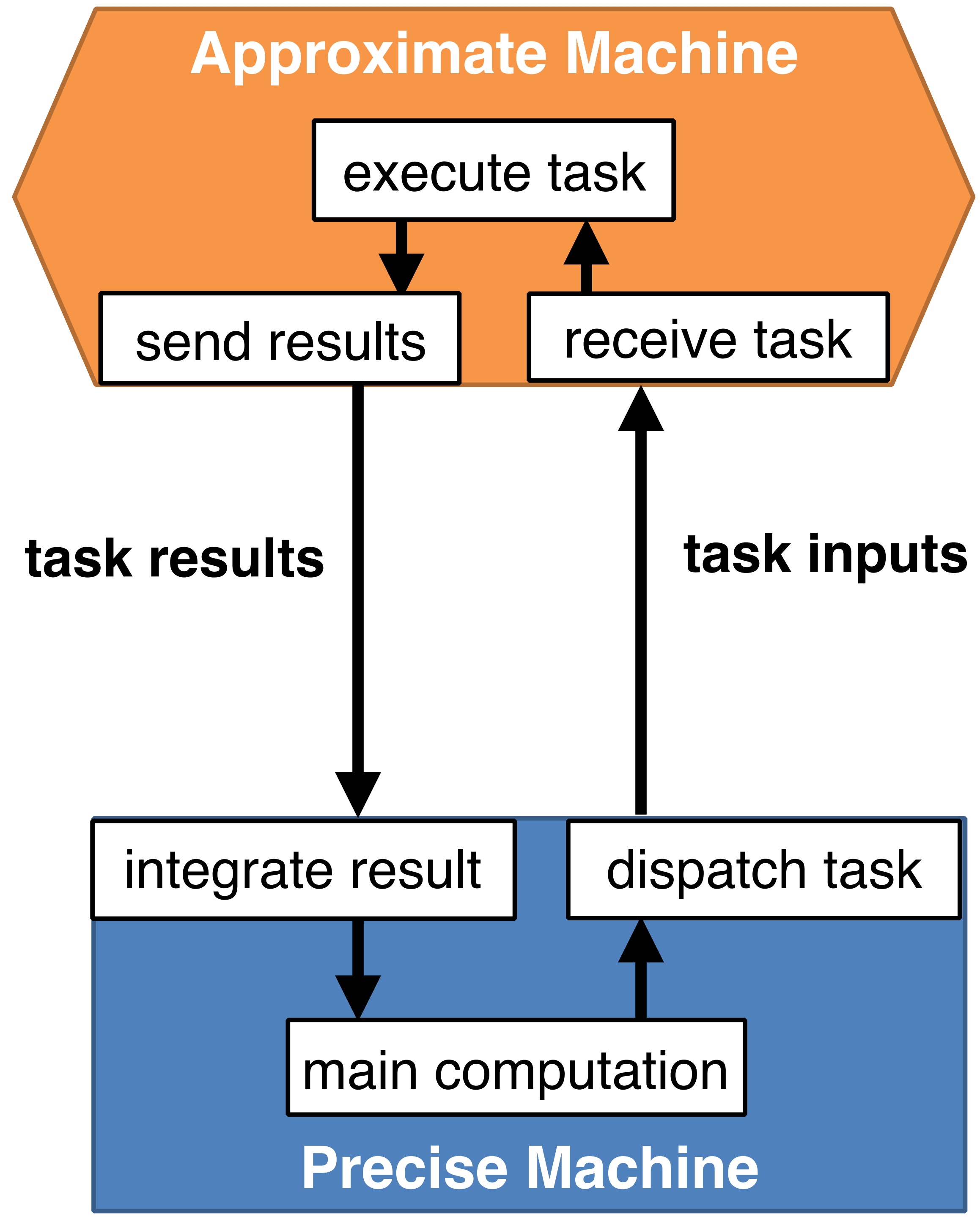
- **Computational Model and Language**
  Ensures computation runs to **completion**

- **Outlier Detection**
  Computation yields an **acceptable** result

- **Optimizations**
  Savings from using approximate hardware

Topaz: Computational Model

- **Task**: approximateable, self contained unit of work
- **Precise** and **approximate** machine
- Precise machine:
  - **Executes** main computation
  - **Dispatch** tasks
  - **Integrates** results into state
  - **Reexecutes** failed tasks
- Approximate machine:
  - **Performs** task computation
  - **returns** a task result



Topaz: Taskset Construct

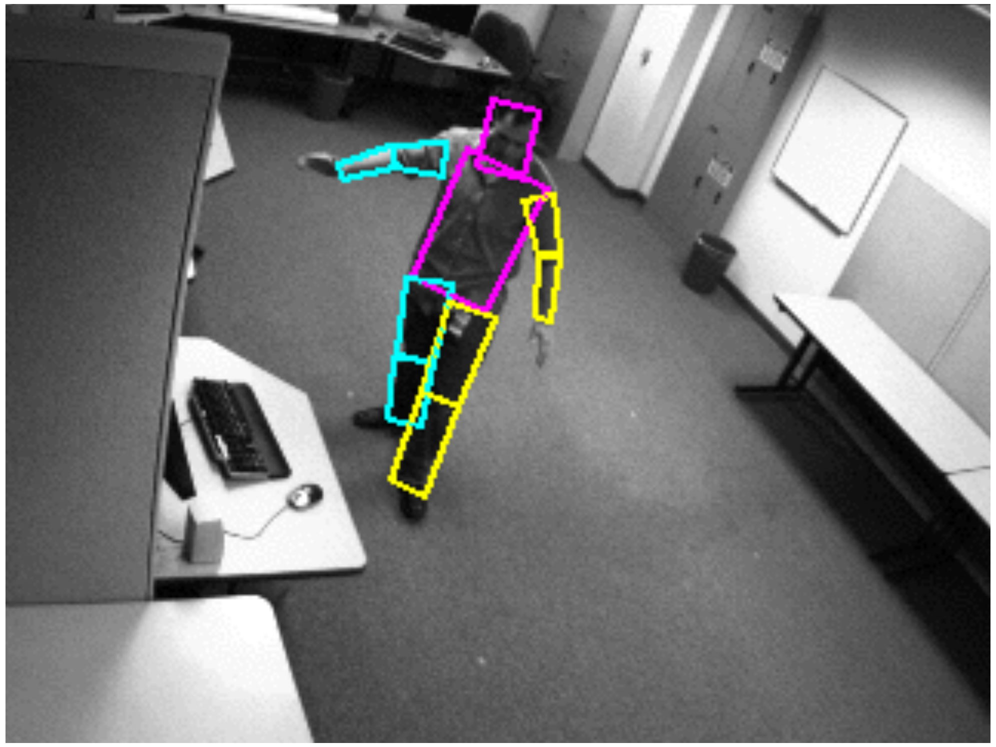
```
taskset  name(int i = l; i < u; i++) {
   compute in  (d1 x1 = e1, ..., dn xn = en)
           out (o1 y1, ..., oj yj) {
     <task body>
   }
   combine { <combine body> }
}
```

- **Taskset**: a set of `u` tasks, where `i` refers to its task
- **Compute**: the task definition. Comprised of `n` inputs and `j` outputs
    - The k^th input: has name `xk`, type `dk` and is assigned to expression `ek`
    - The k^th result: has name `vk`, type `ok`
    - The `<task body>` describes the task computation
- **Combine**: the routine for integrating the task into the main computation.

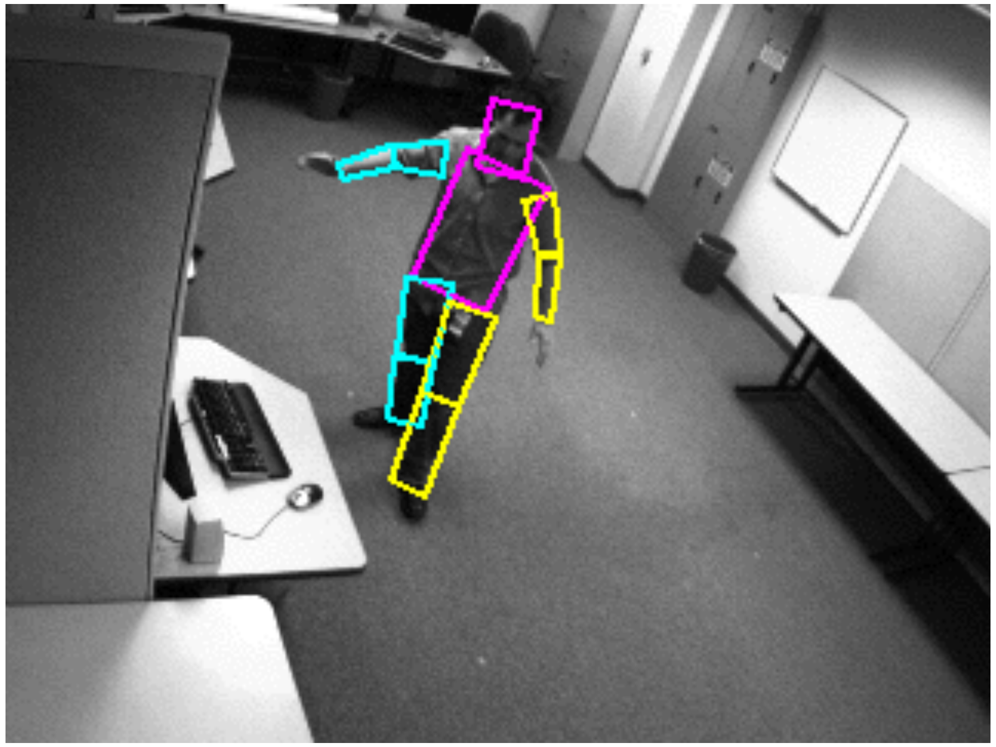
Topaz: Bodytrack Example

- Computation: find the pose that best fits the person in each camera frame

- Target Routine: compute the weight for all poses, given an image
  - Task: compute the weight of one pose, given image
  - Integration: write computed weight to global weight array

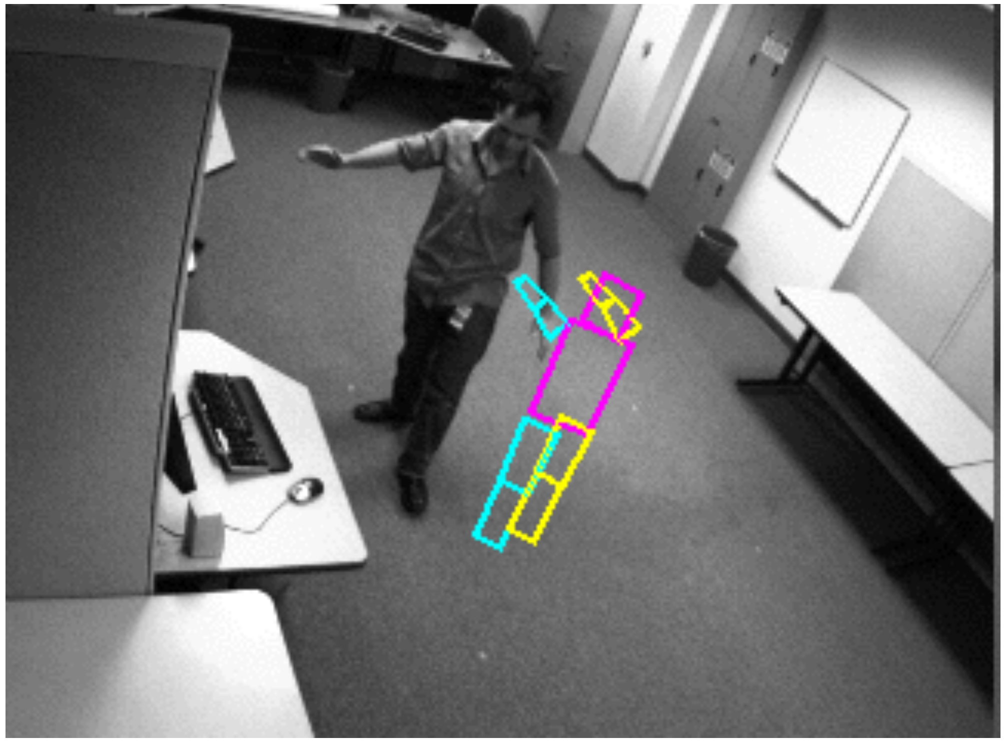


Topaz: Bodytrack Example



Pose is good fit

weight = 1.03



Pose is bad fit

weight = 0.12

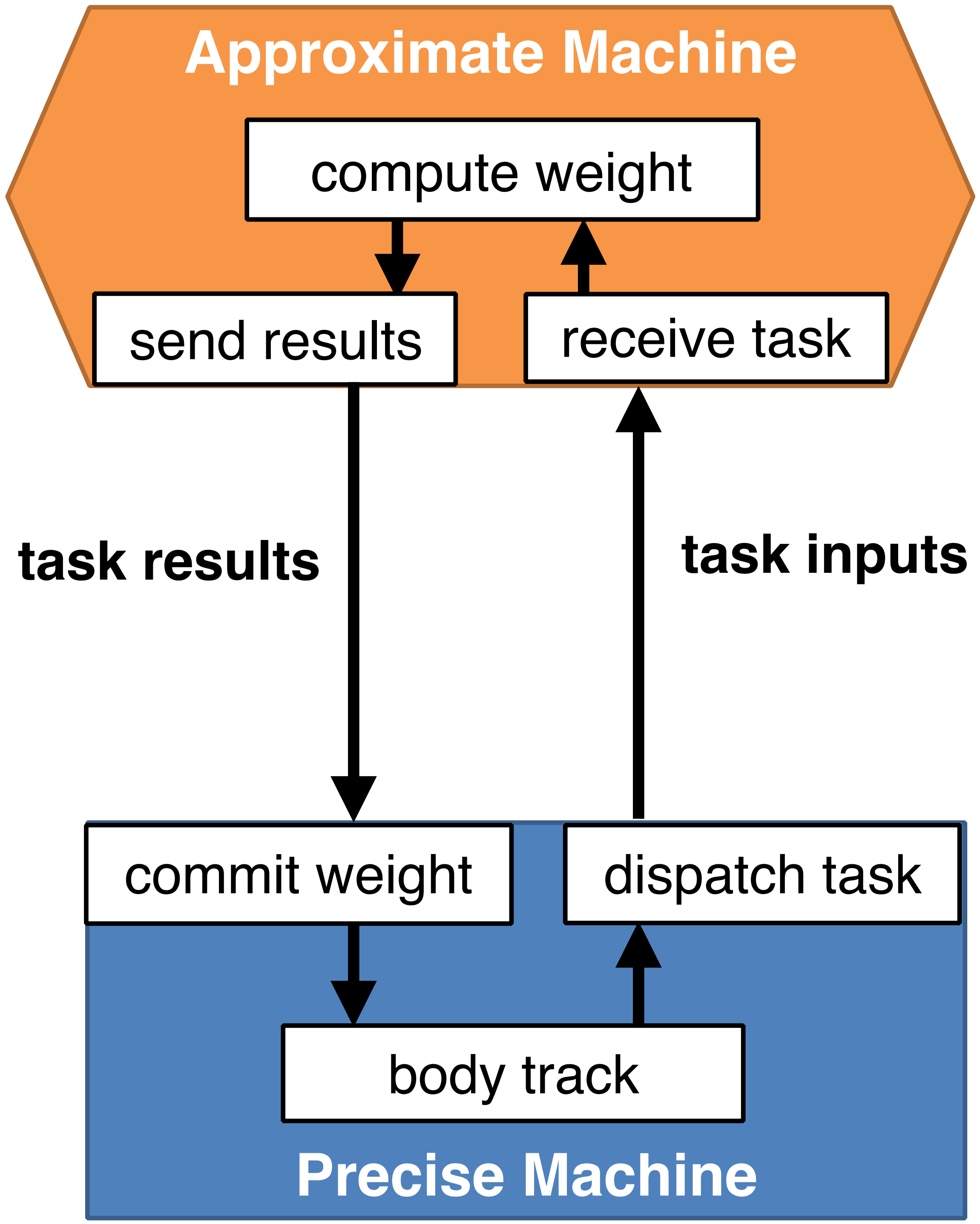
Topaz: Bodytrack Example

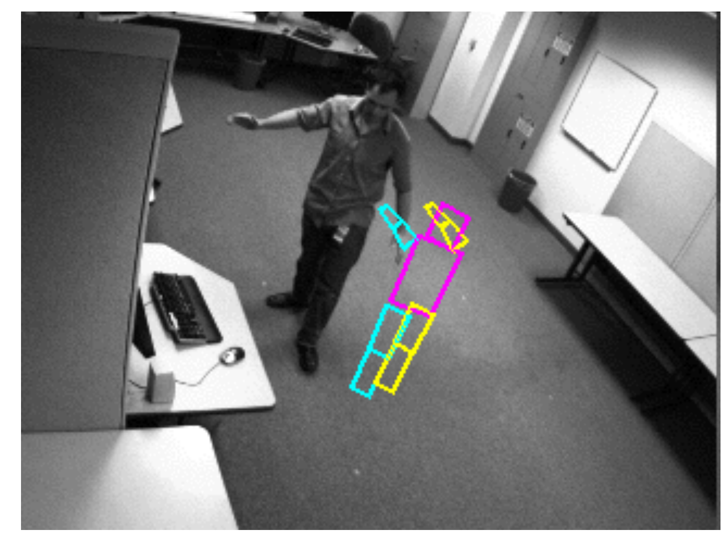
```
// computes the weights for each valid pose.
taskset   calcweights(i=0; i<particles.size(); i+=1){
  compute in (
     float tpart[P_SIZE] =  (float*) particles[i],
     float tmodel[M_SIZE] = (float*) mdl_prim,
     char timg[I_SIZE] = (char *) img_prim,
     int nCams = mModel->NCameras(),
     int nBits = mModel->getBytesPerPixel(),
     int width = mModel->getWidth(),
     int height =mModel->getHeight()
  ) out (float tweight) {

      tweight = CalcWeight(tpart,
           tmodel, timg, nCams, width, height, nBits);

  } combine {
      mWeights[i] = tweight;
  }
}
```

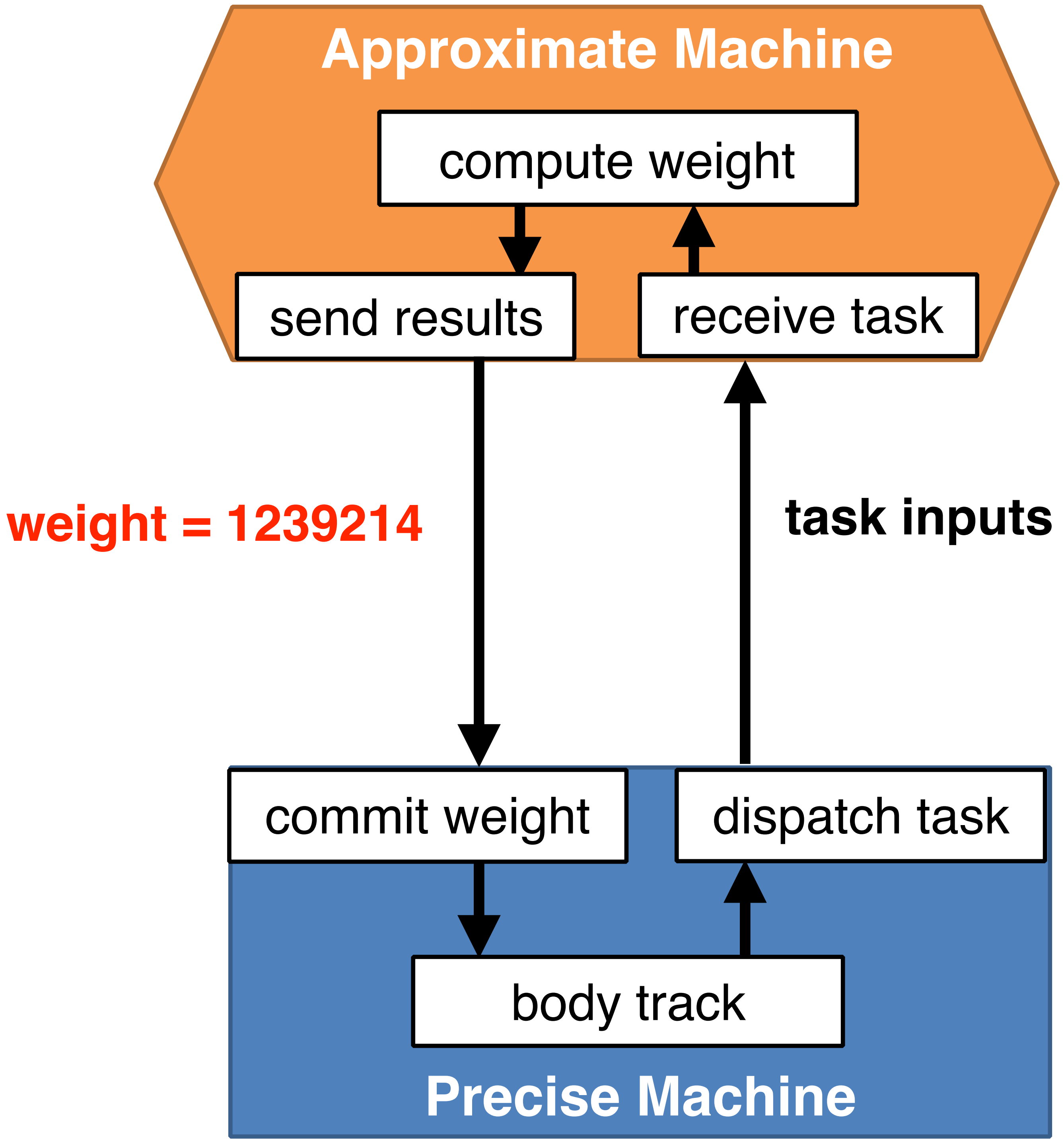
Topaz: Bodytrack Example

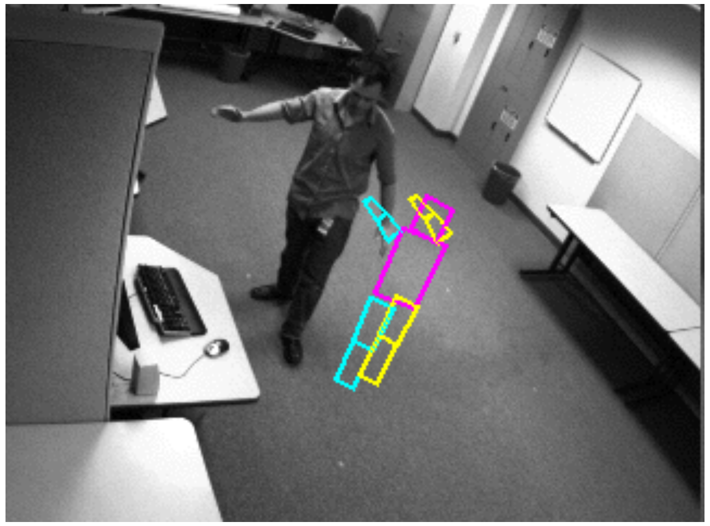




weight = 0.12

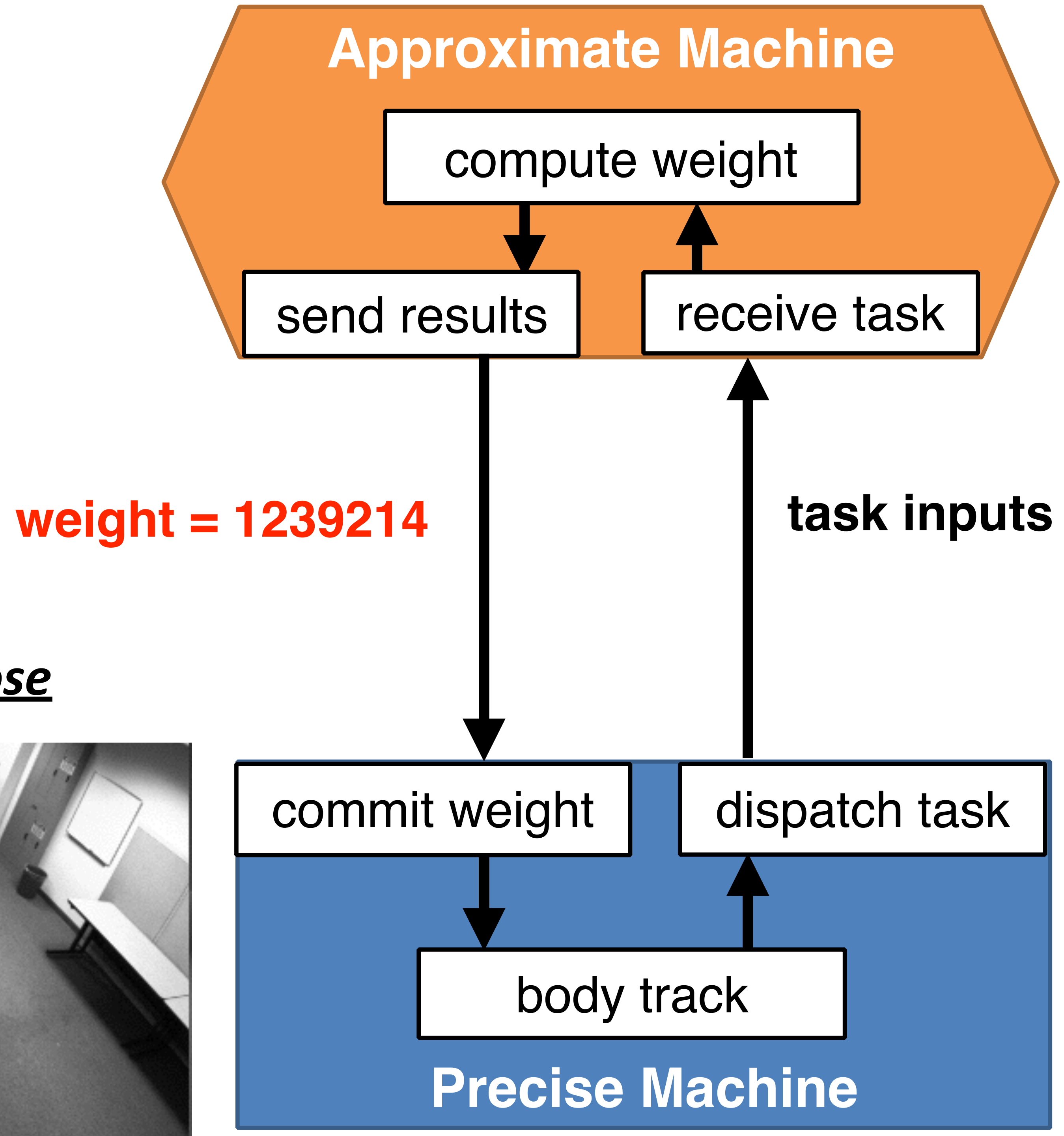
Topaz: Bodytrack Example



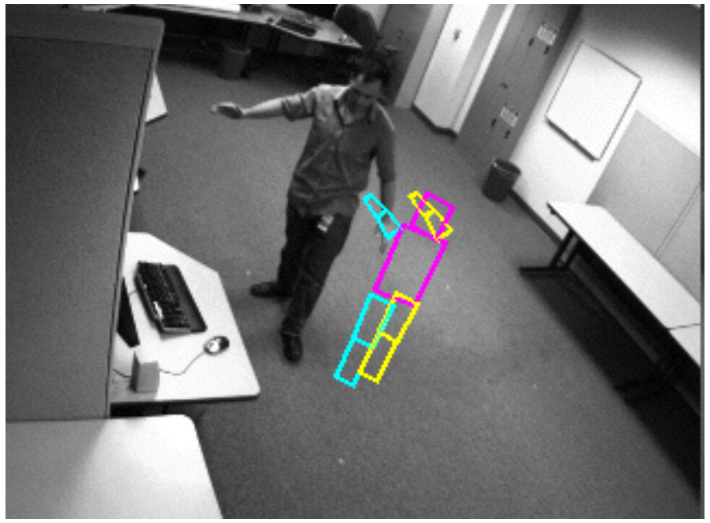


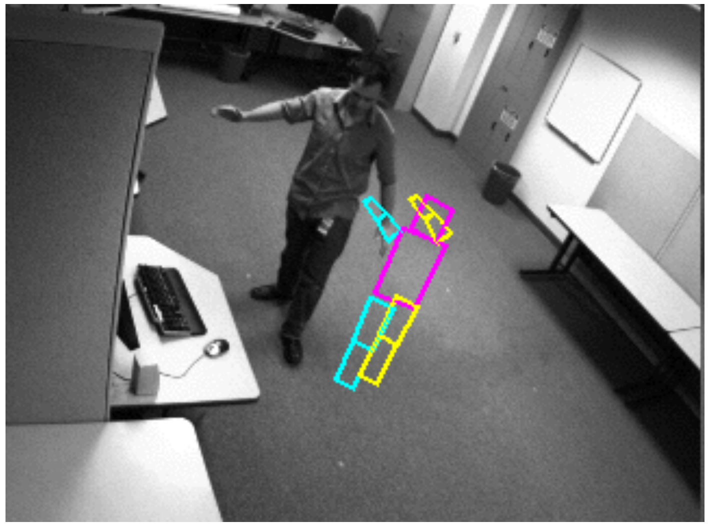
weight = 0.12

Topaz: Bodytrack Example



Best Pose

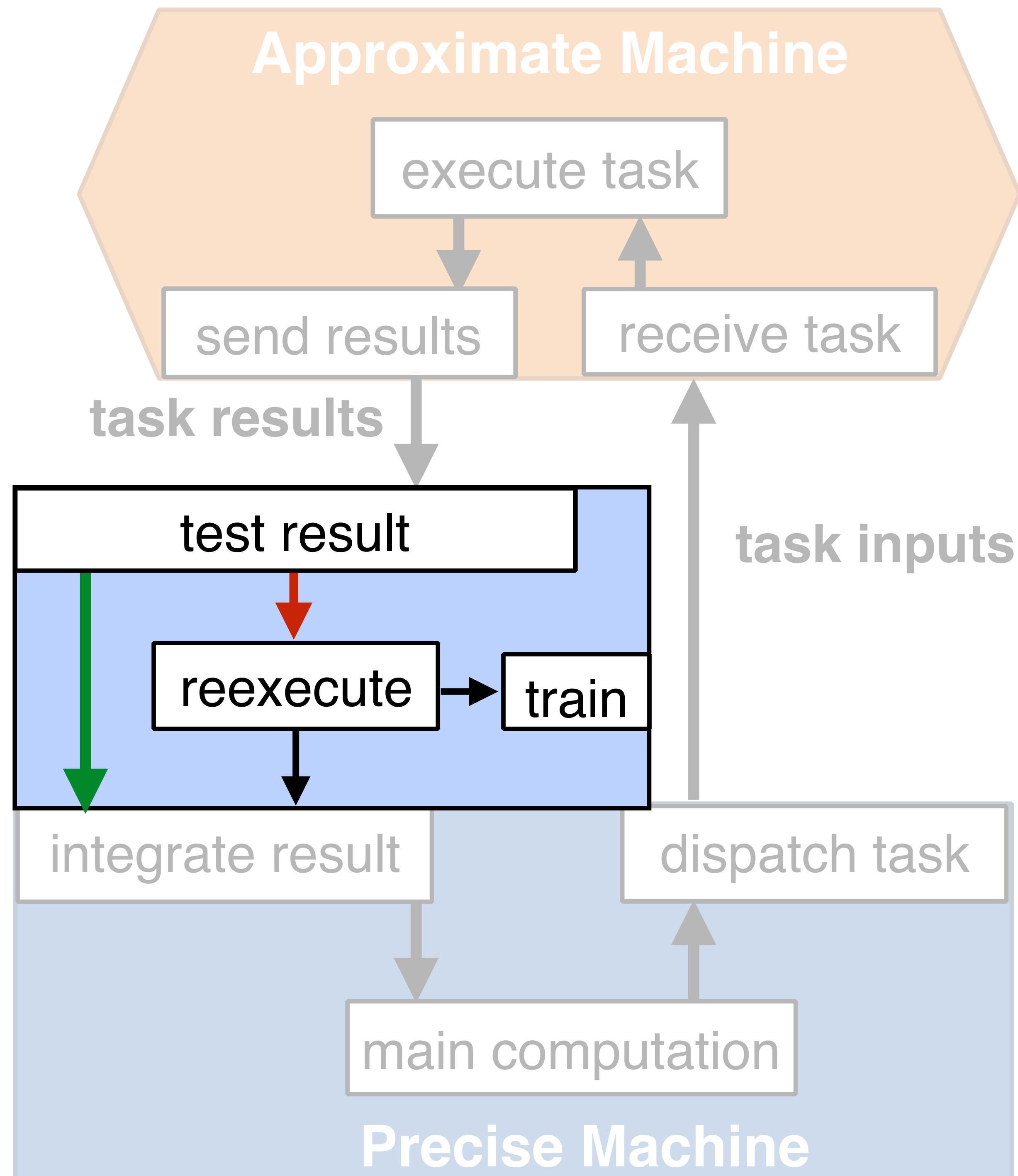




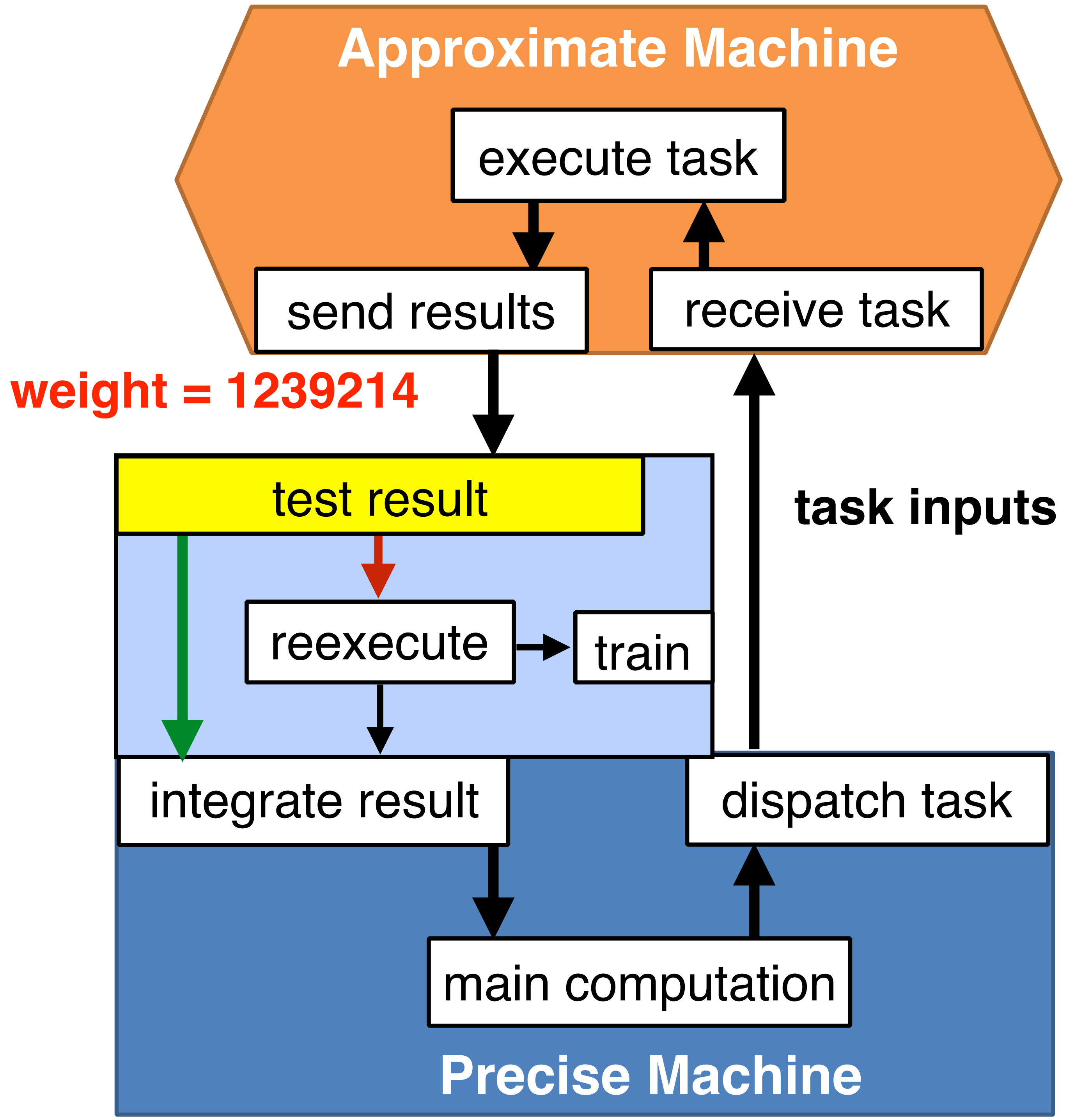
weight = 0.12

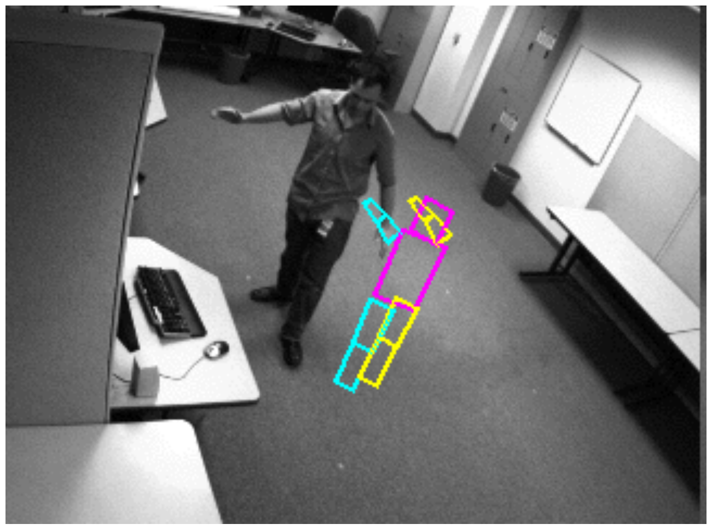
Basic Outlier Detection: Overview

- **Outlier Detection**
- **Precise** machine performs outlier detection on result tuple
  - On **accept**:
    - **Integrate** task result
  - On **reject**:
    - **Reexecute** task on reliable hardware
    - **Train** outlier detector
    - **Integrate** task result

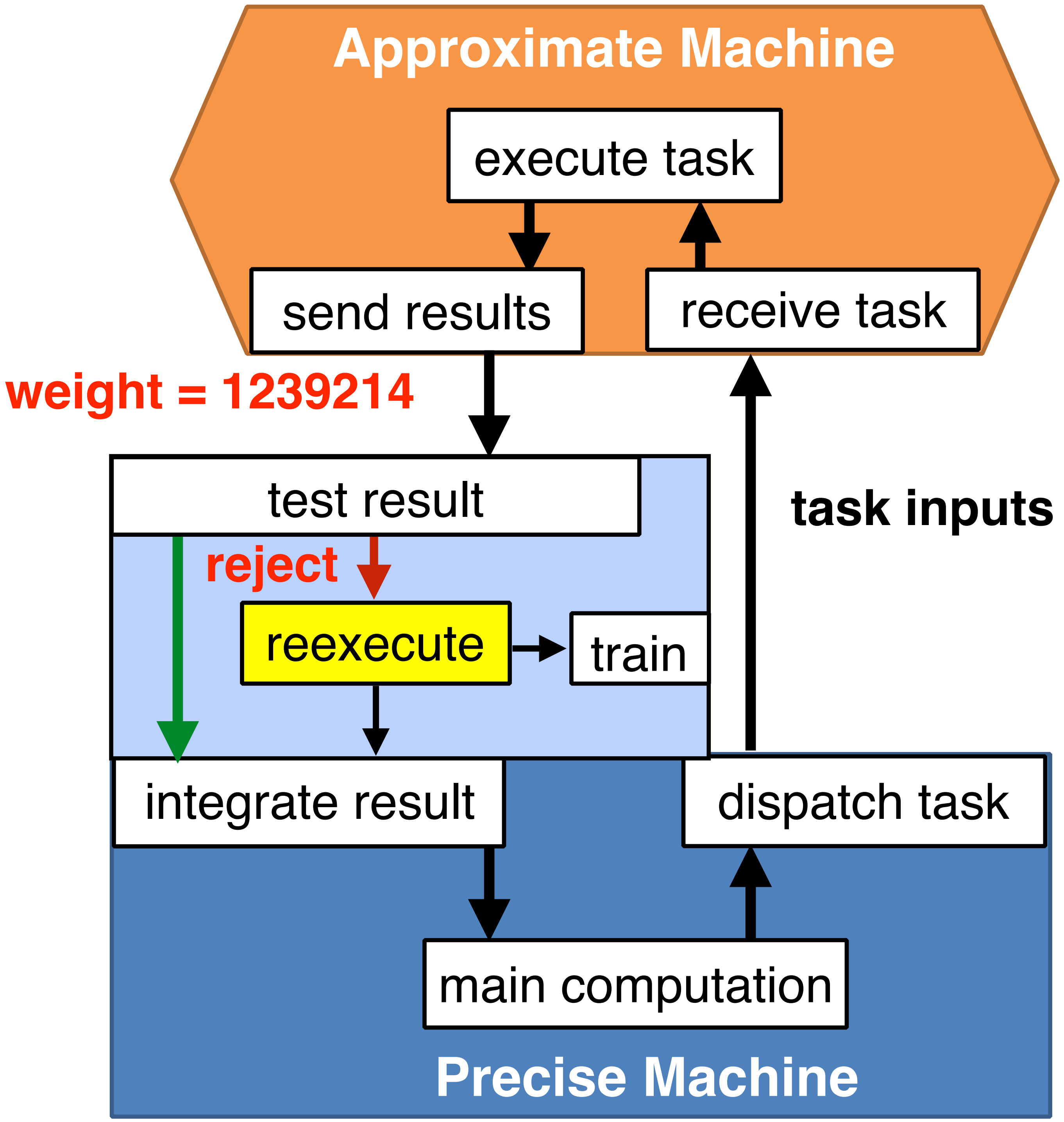


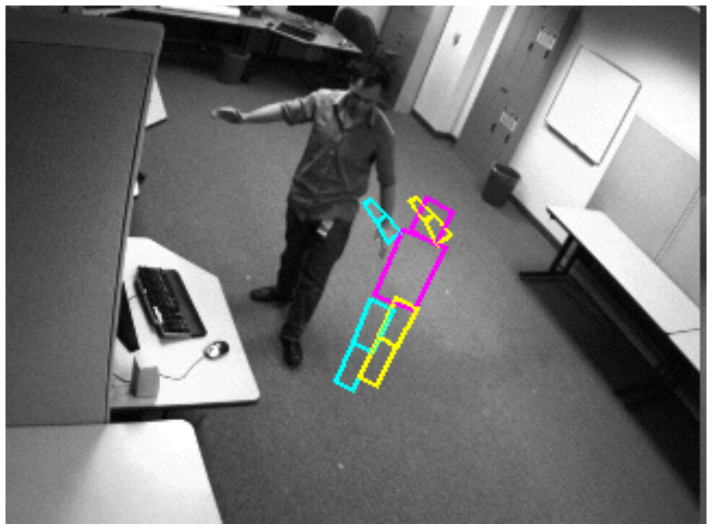
Topaz: Bodytrack Example



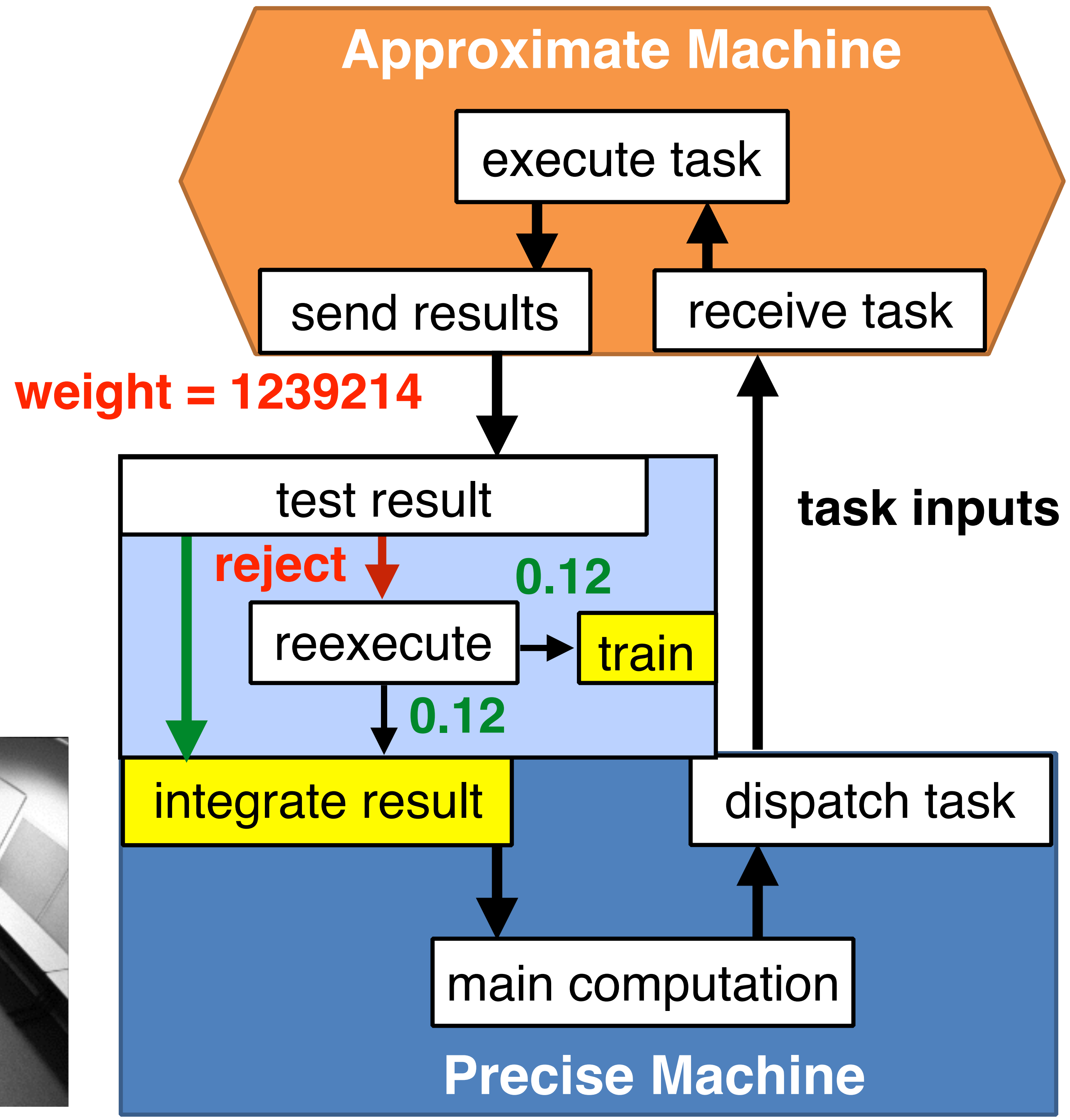


Topaz: Bodytrack Example

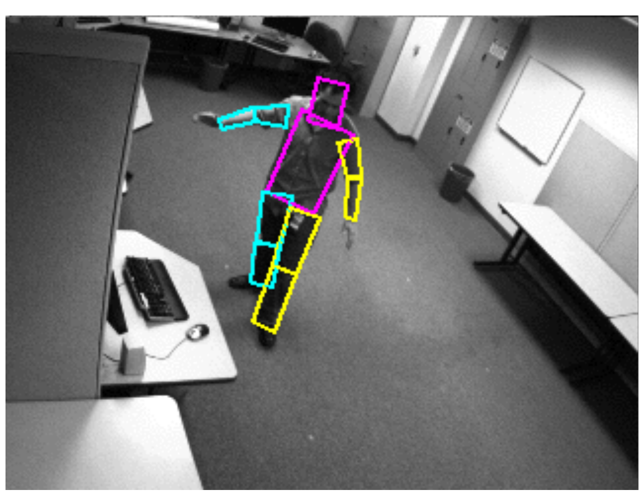


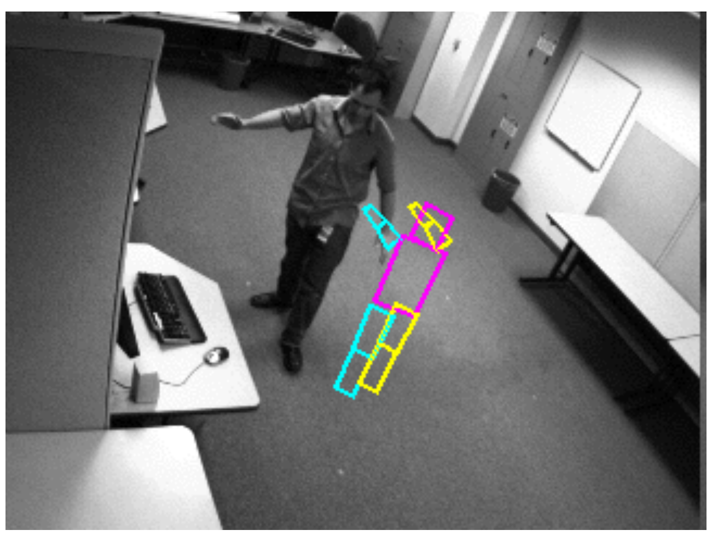


Topaz: Bodytrack Example



Best Pose





Testing results using outlier detection

- **<u>Algorithm</u>**

  Given result tuple of n elements

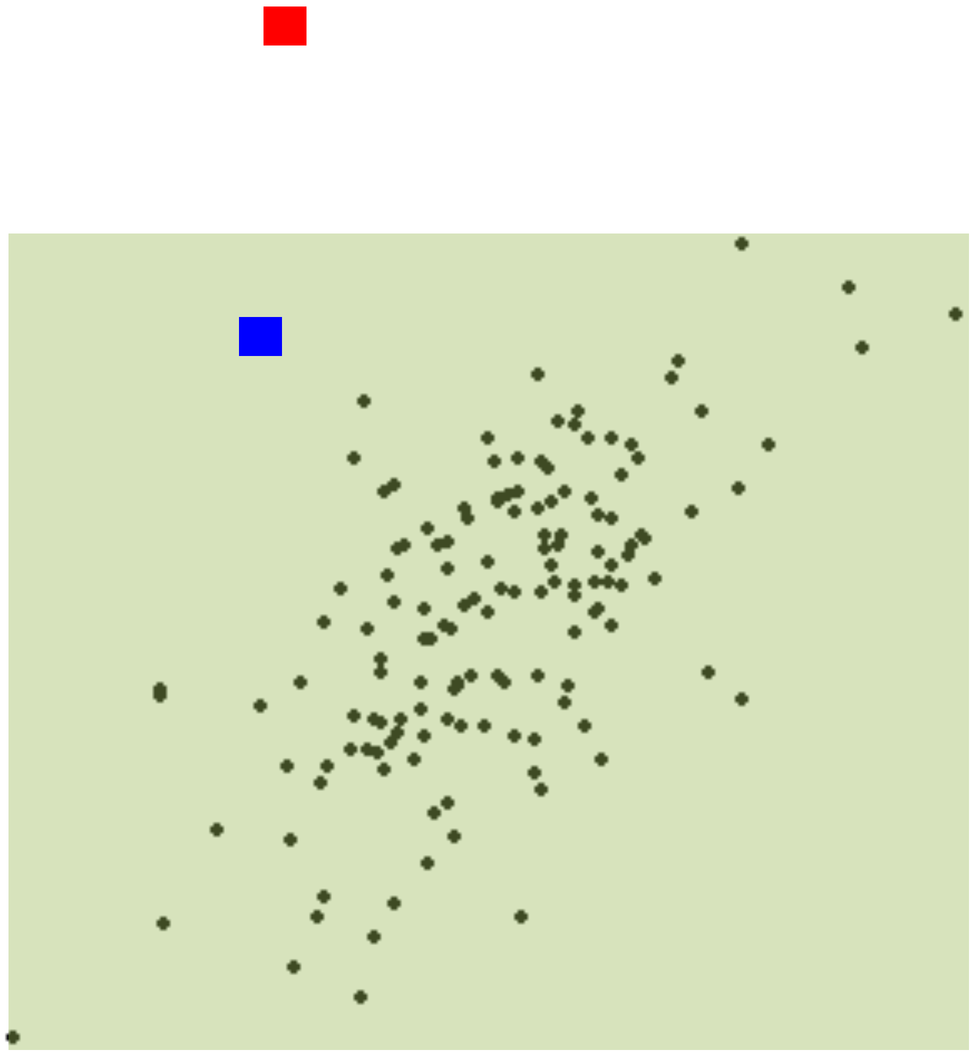
  In acceptance region:

  Accept

  Otherwise:

  Reject

- **Model: Acceptance Region**
  - N-dim hyper rectangle
  - Result tuple distribution
  - Dim: min, max of an element



Training the outlier detector

- **The Training Process**
  - Online learning
  - “Learn from failure”
  - Use reexecuted task results

- **Algorithm**

If test(x) = reject:

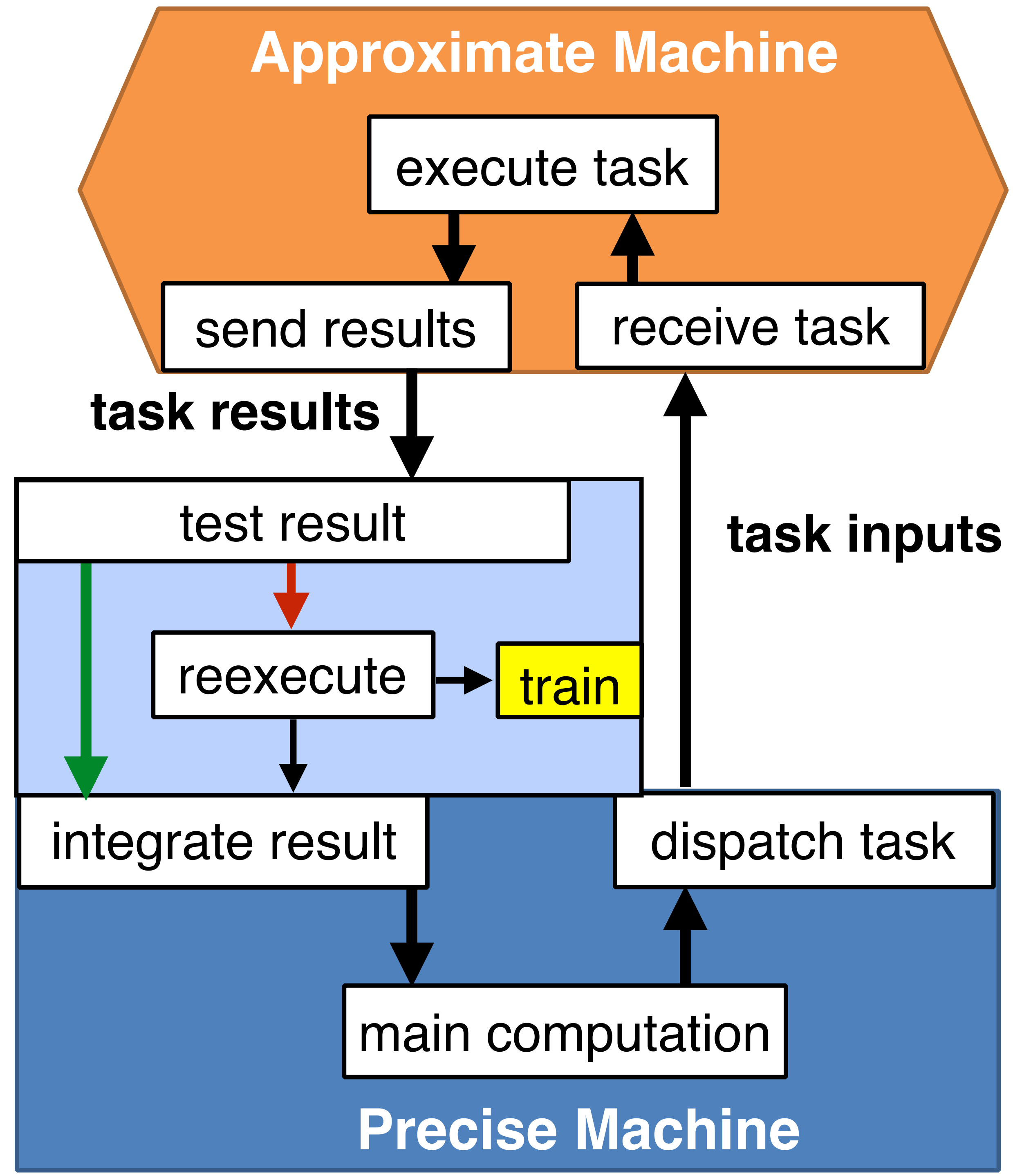
Reexecute task to obtain x’

Update r to include x’

Integrate x’

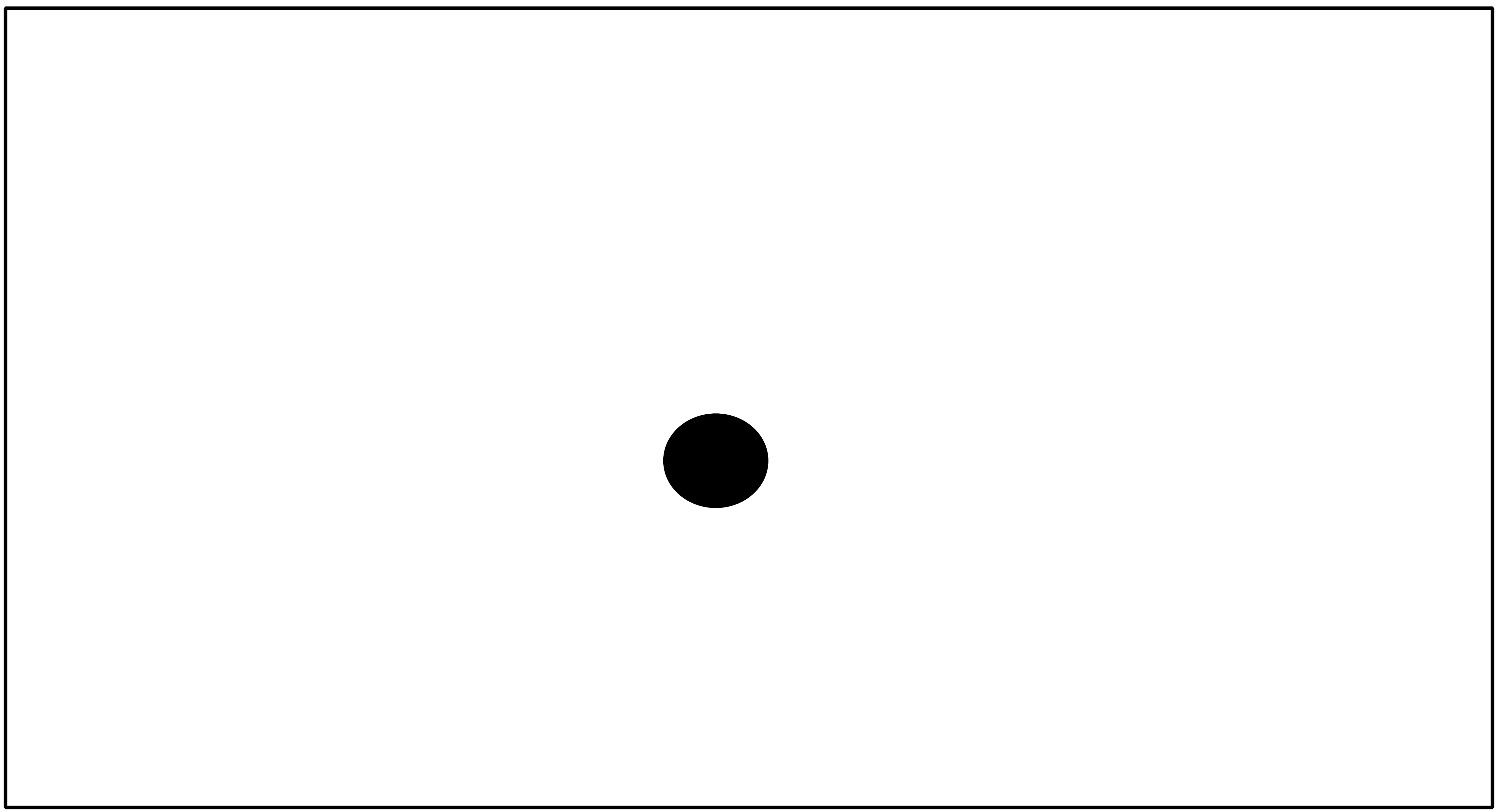
Otherwise:

Integrate x



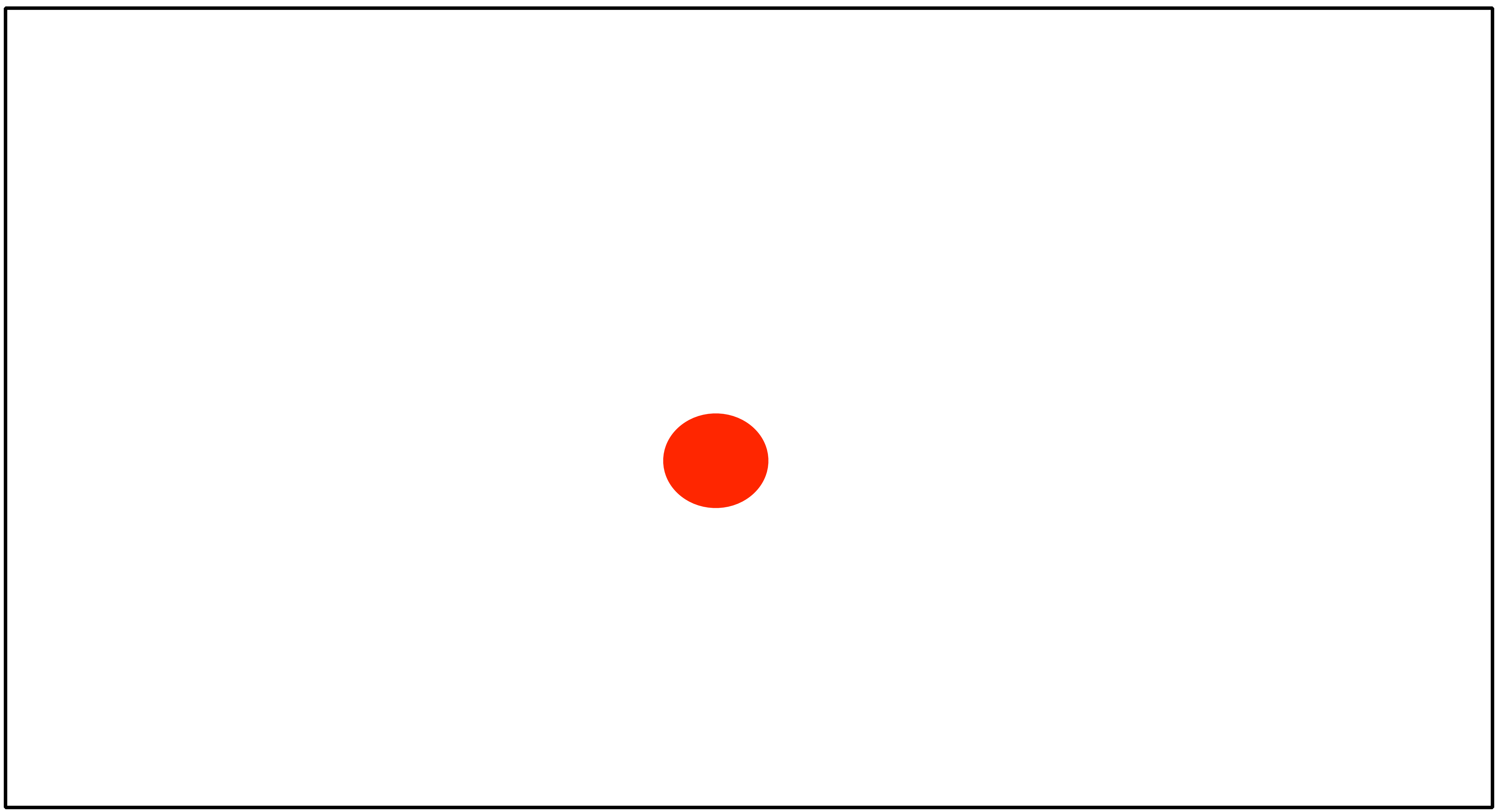
Outlier detector: an example

Receive Task Result for Task 1



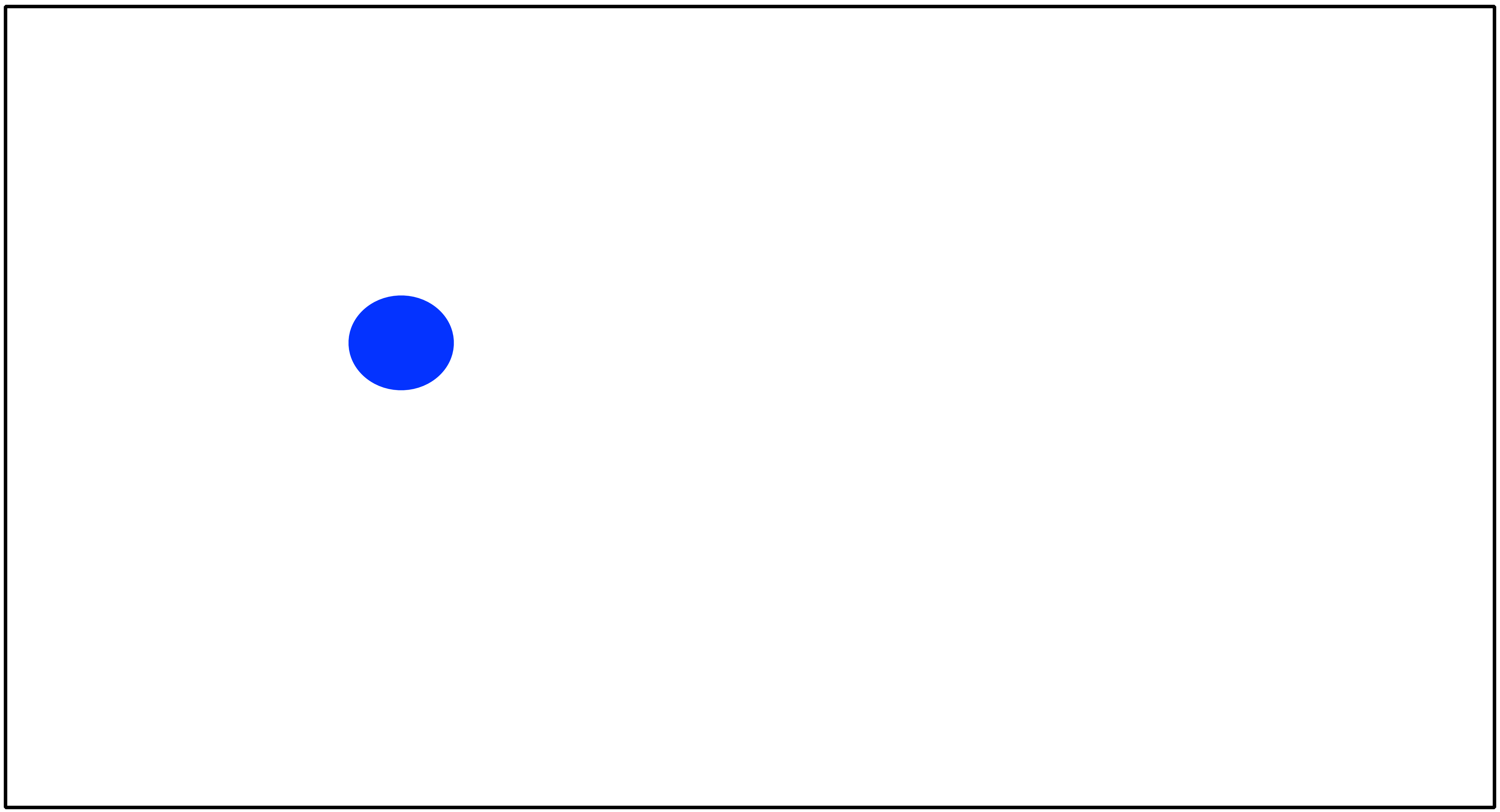
Outlier detector: an example

Reject task result for task 1



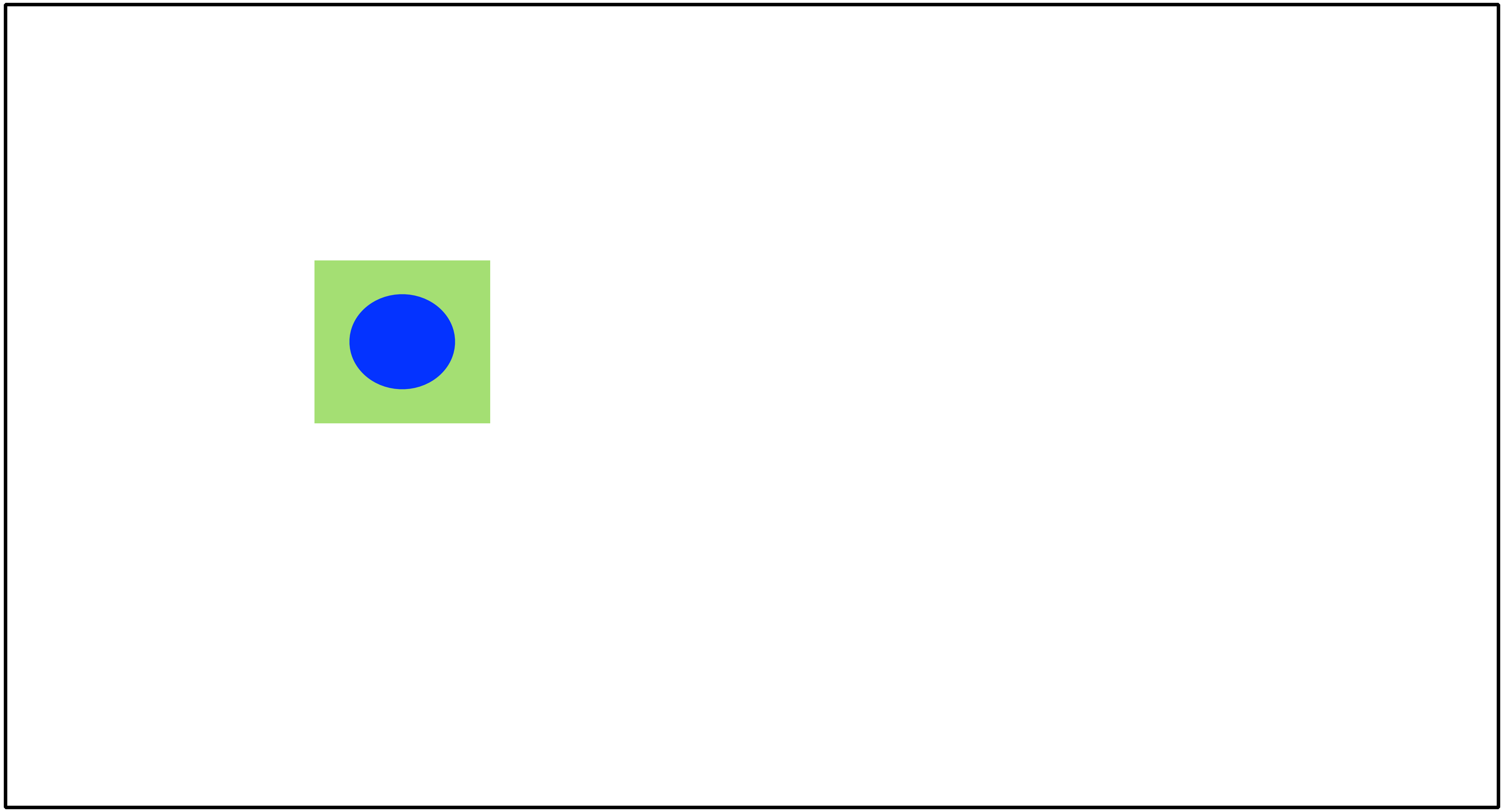
Outlier detector: an example

Reexecute task result for task 1



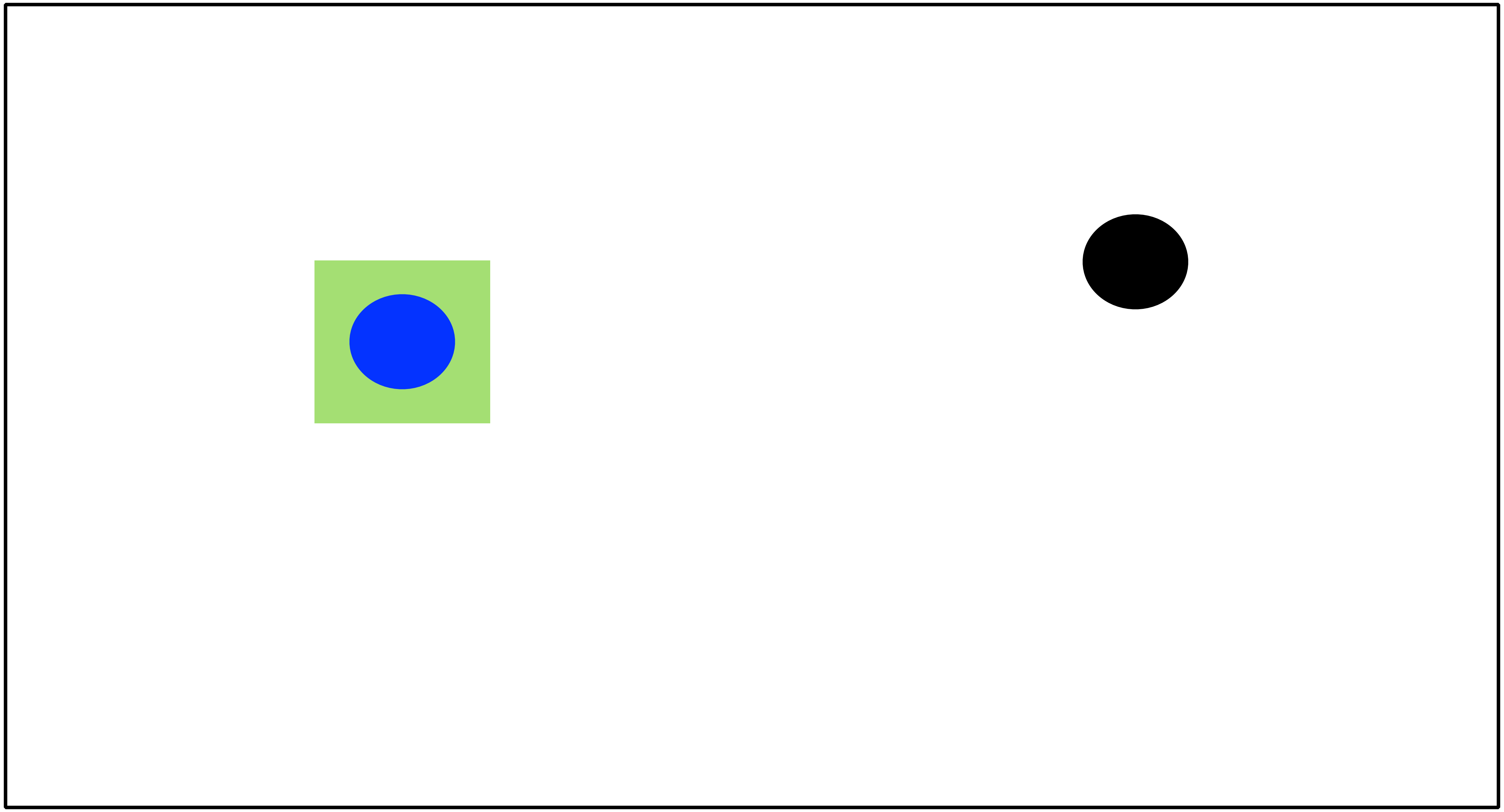
Outlier detector: an example

Training: Expand acceptance region to include result for task 1



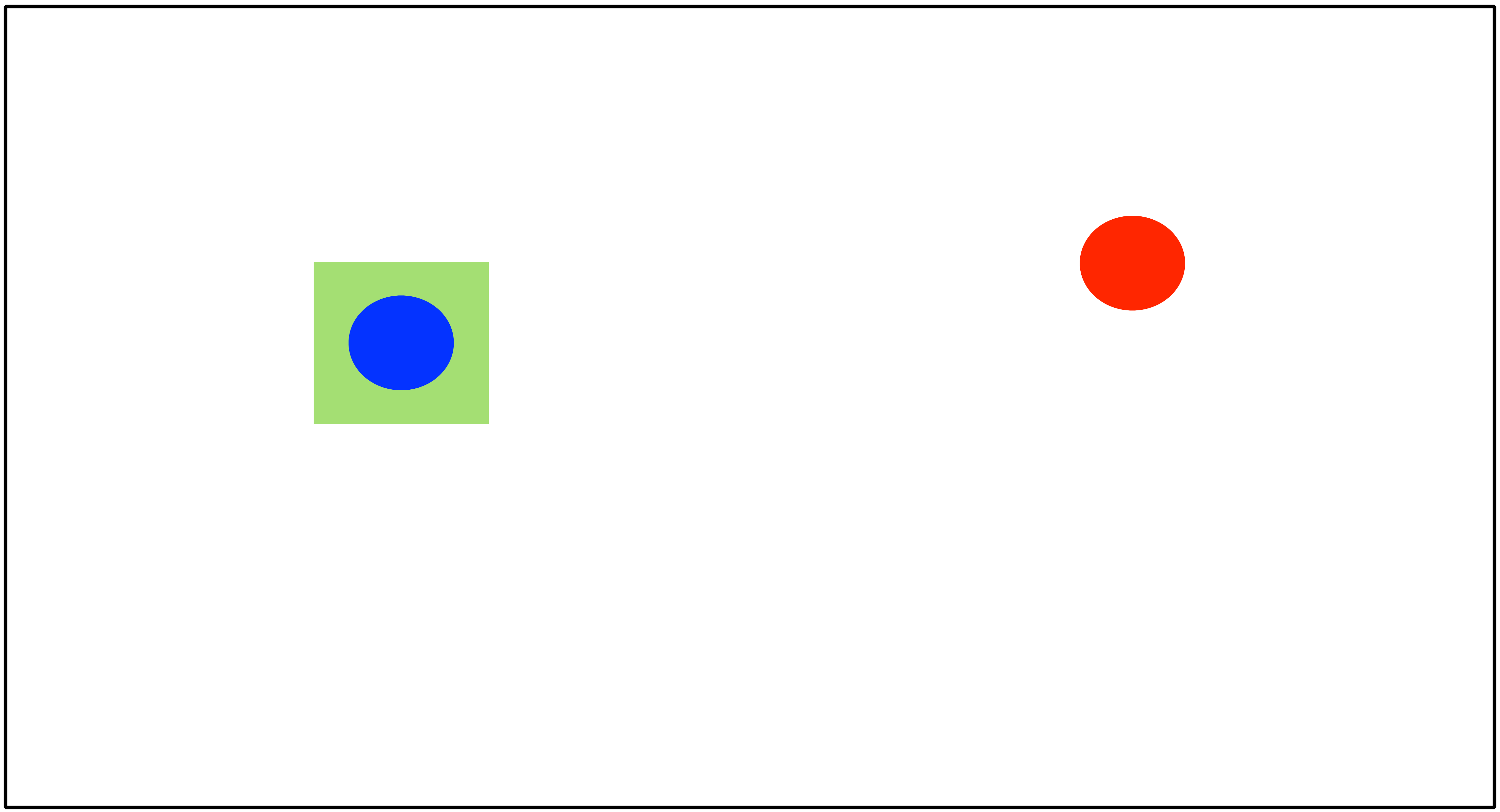
Outlier detector: an example

Receive task result for task 2



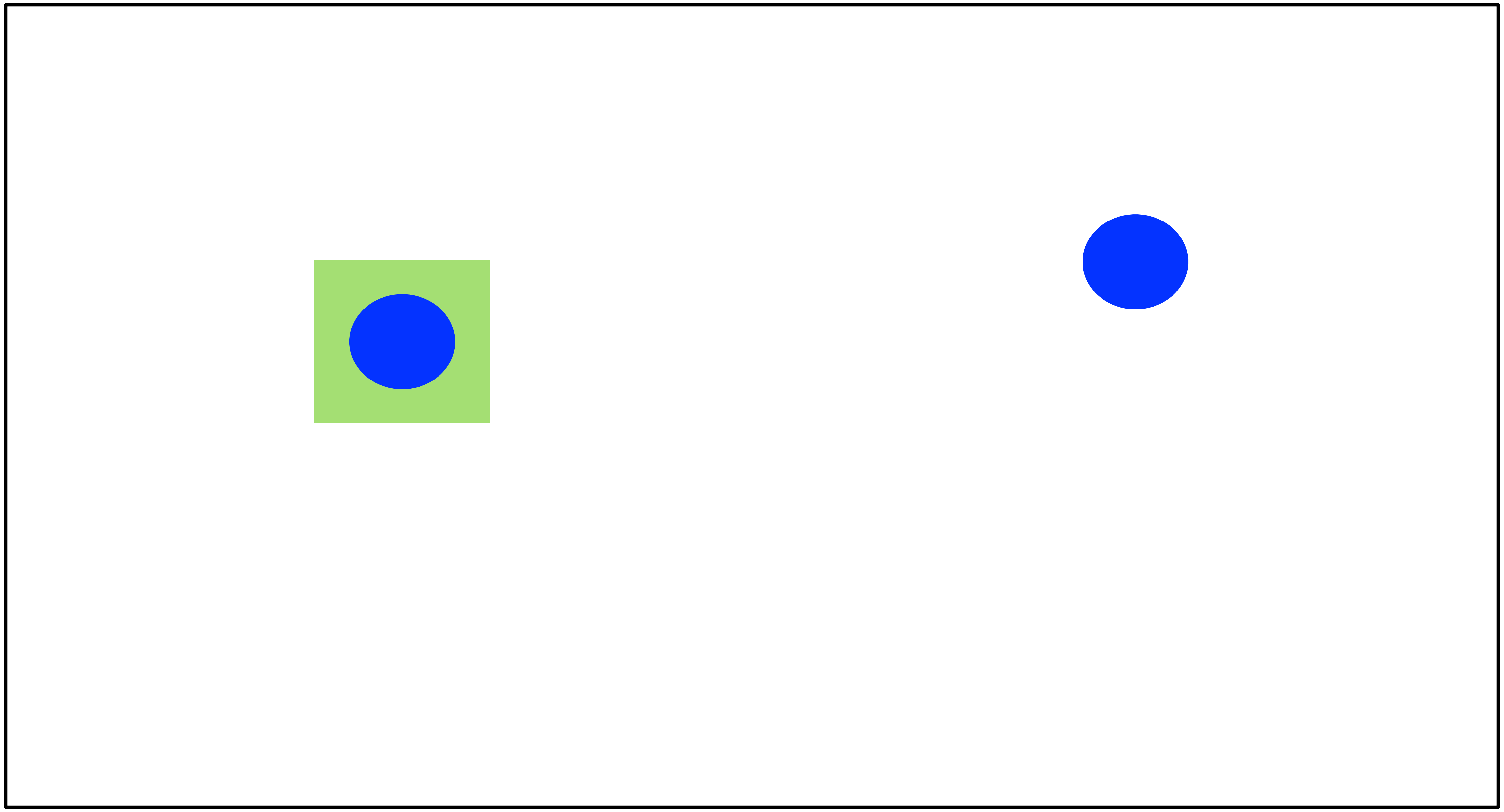
Outlier detector: an example

Reject task result for task 2



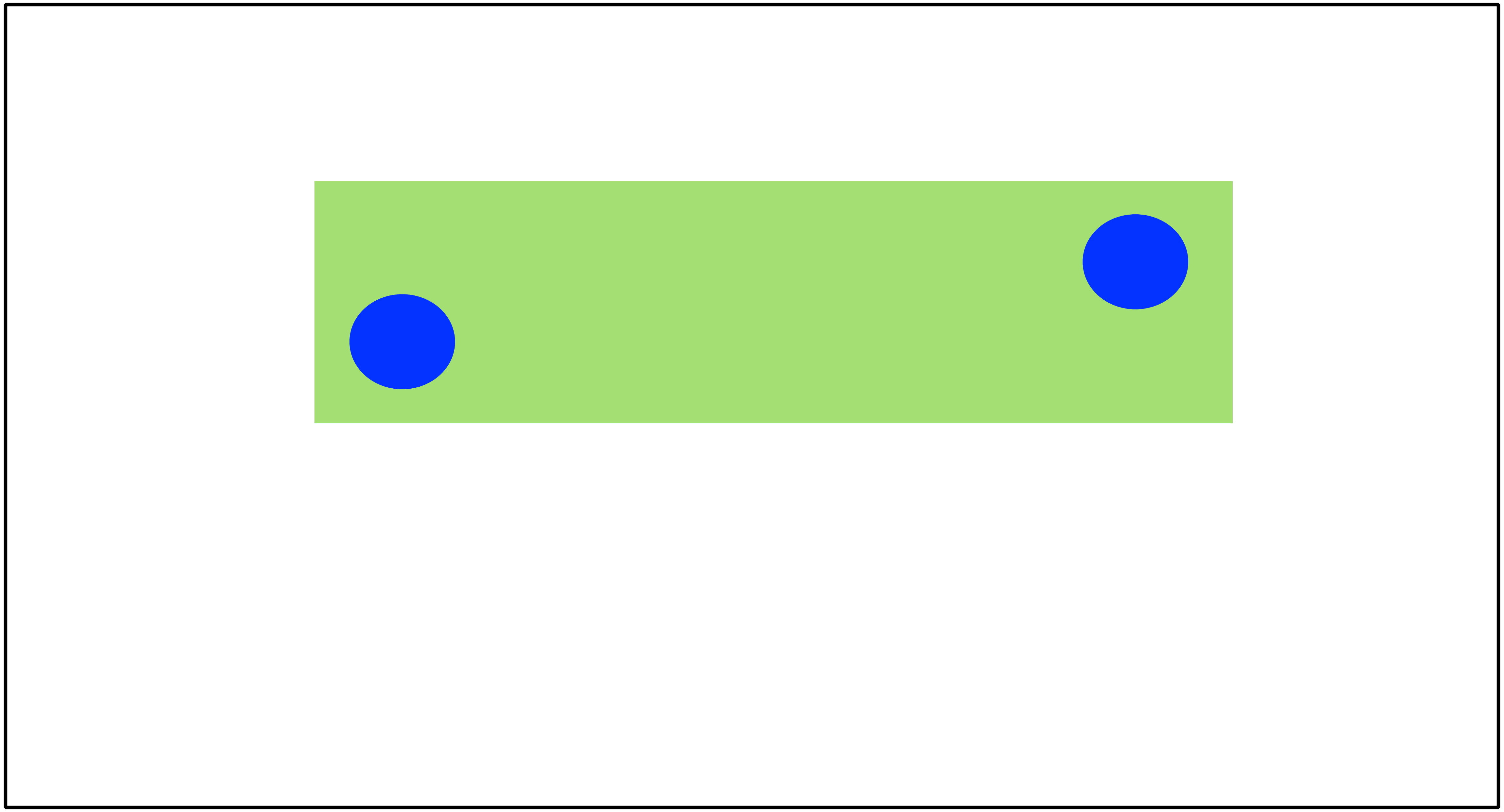
Outlier detector: an example

Reexecute task result for task 2



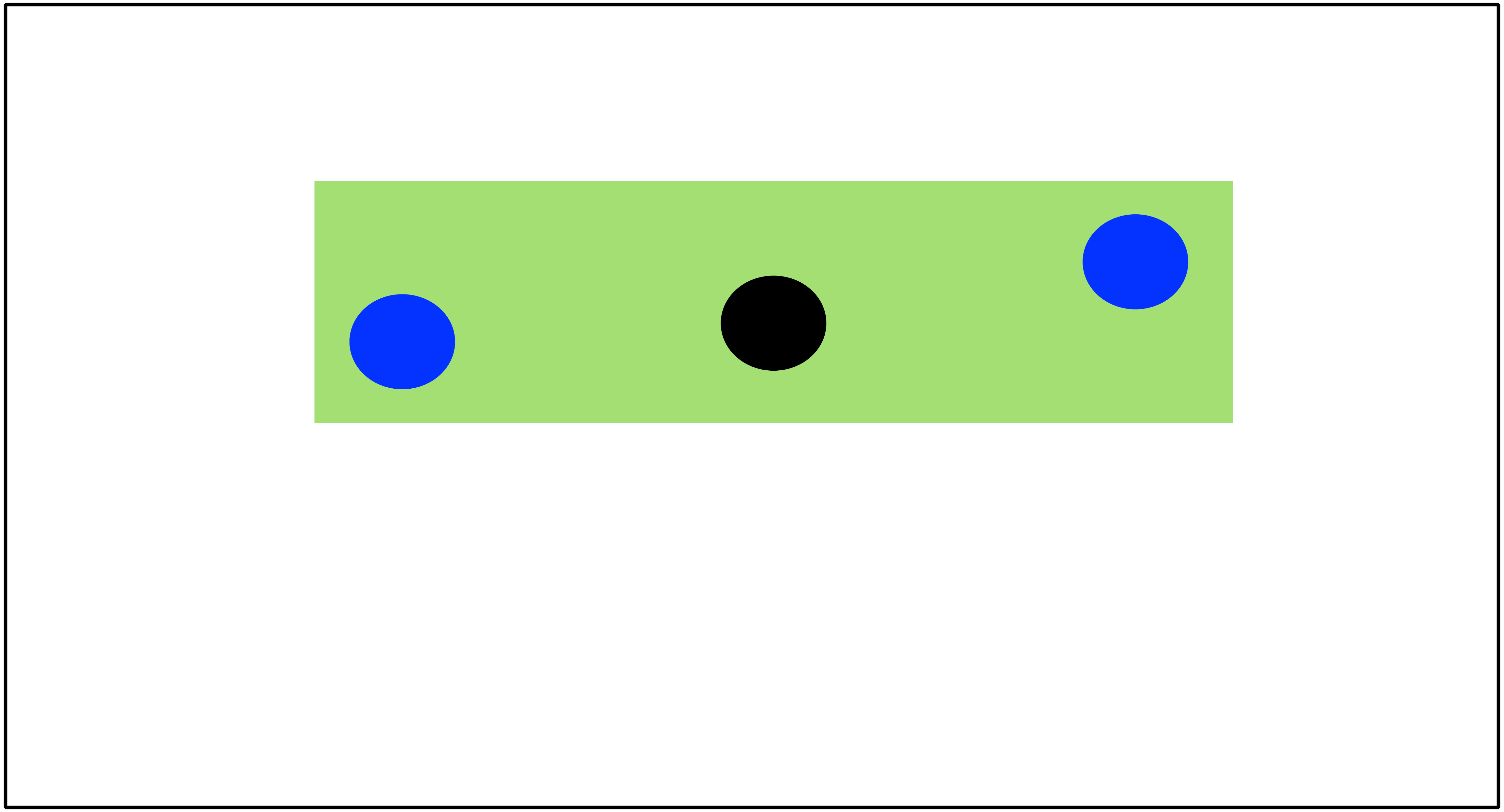
Outlier detector: an example

Training: expand acceptance region to include result for task 2



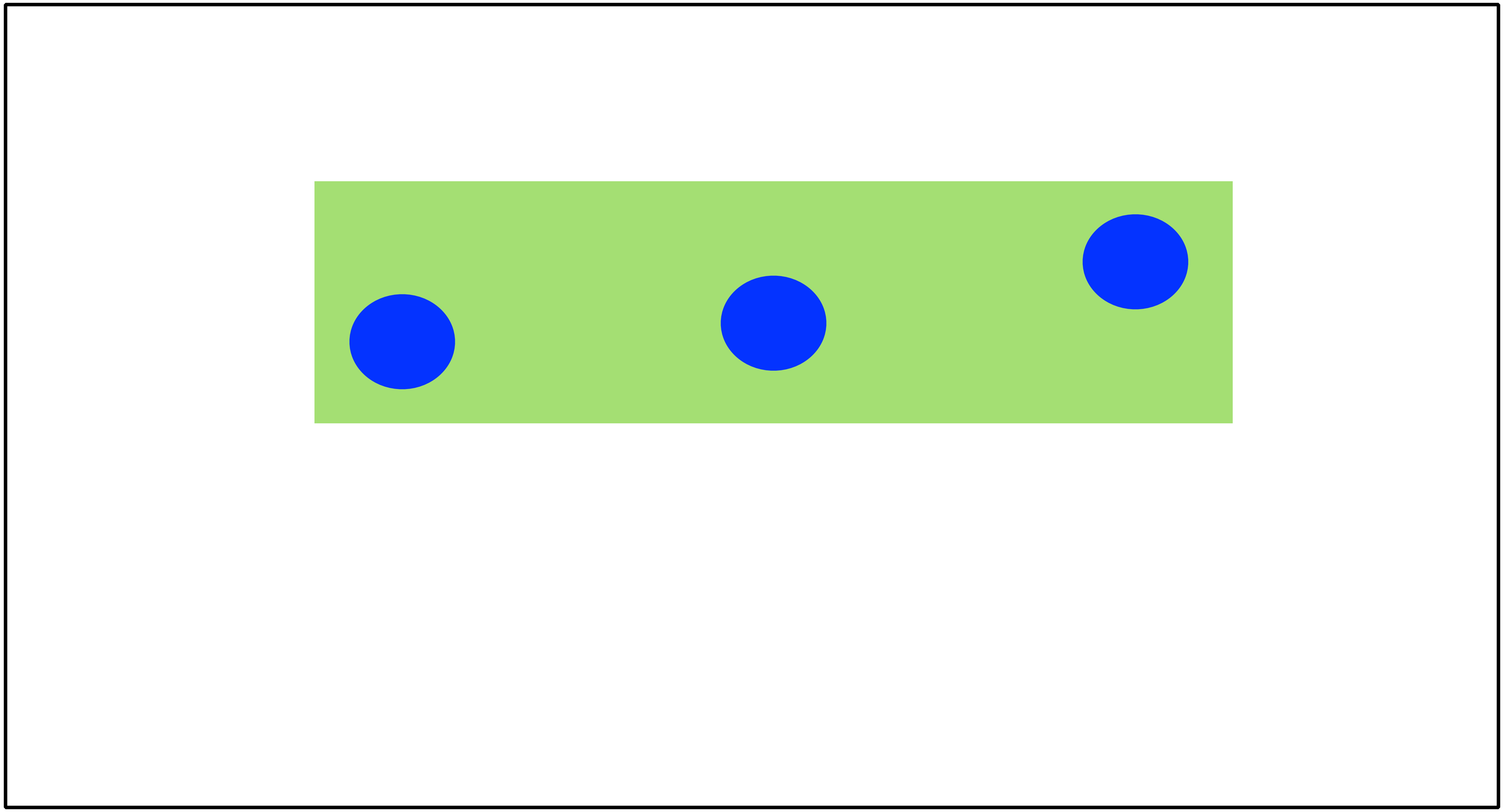
Outlier detector: an example

Receive task result for task 3



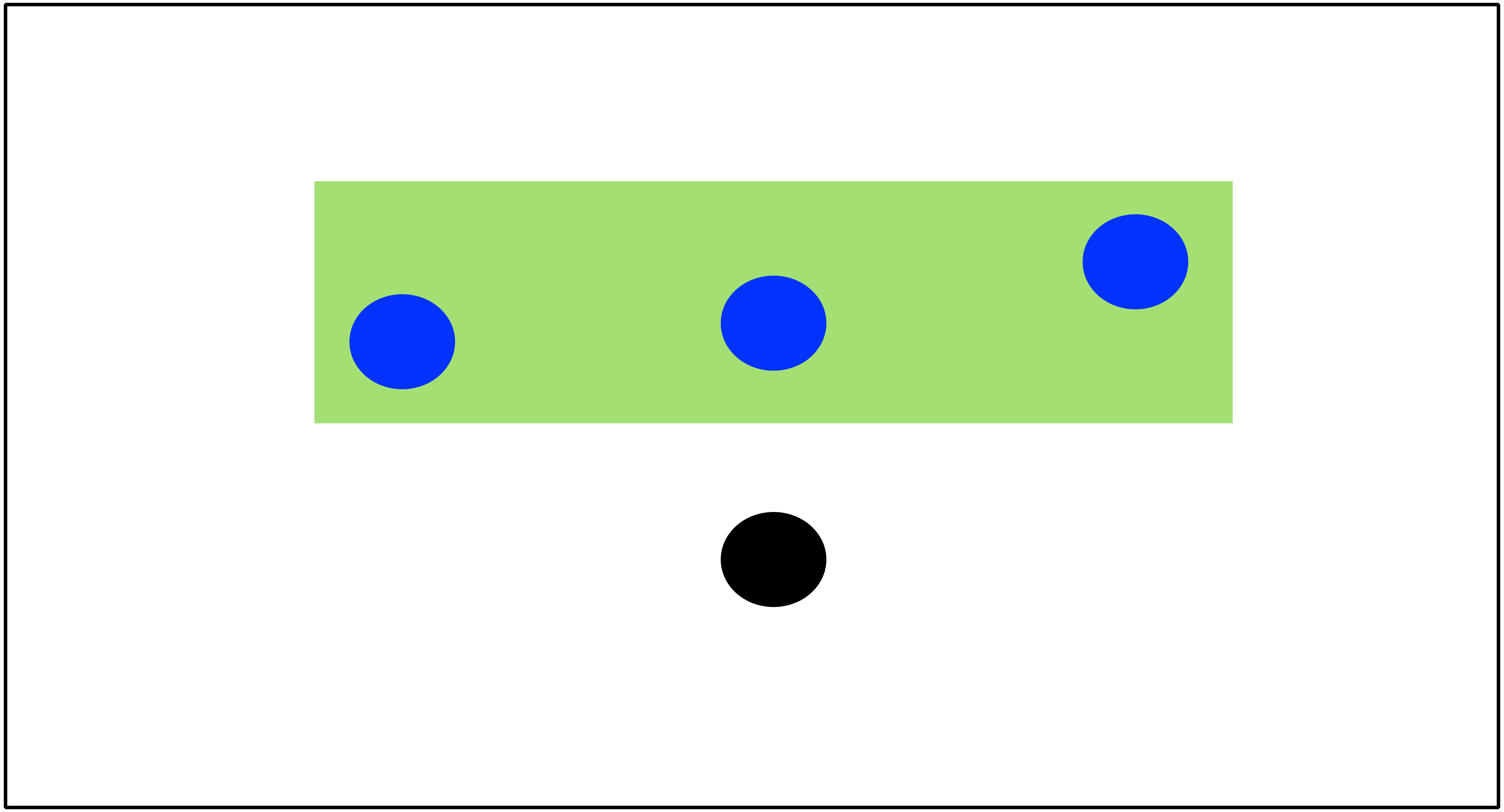
Outlier detector: an example

Accept task result for task 3



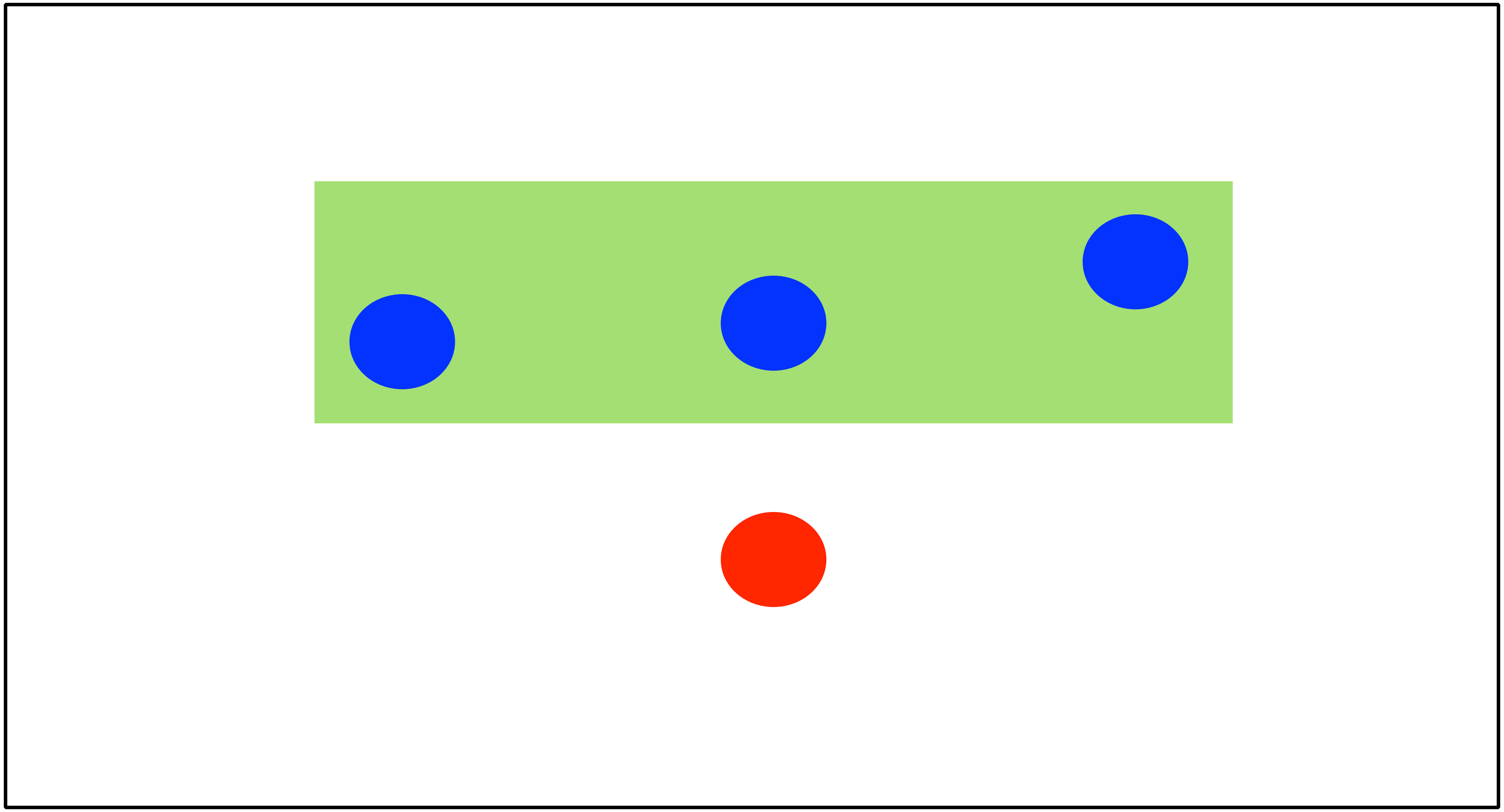
Outlier detector: an example

Receive task result for task 4



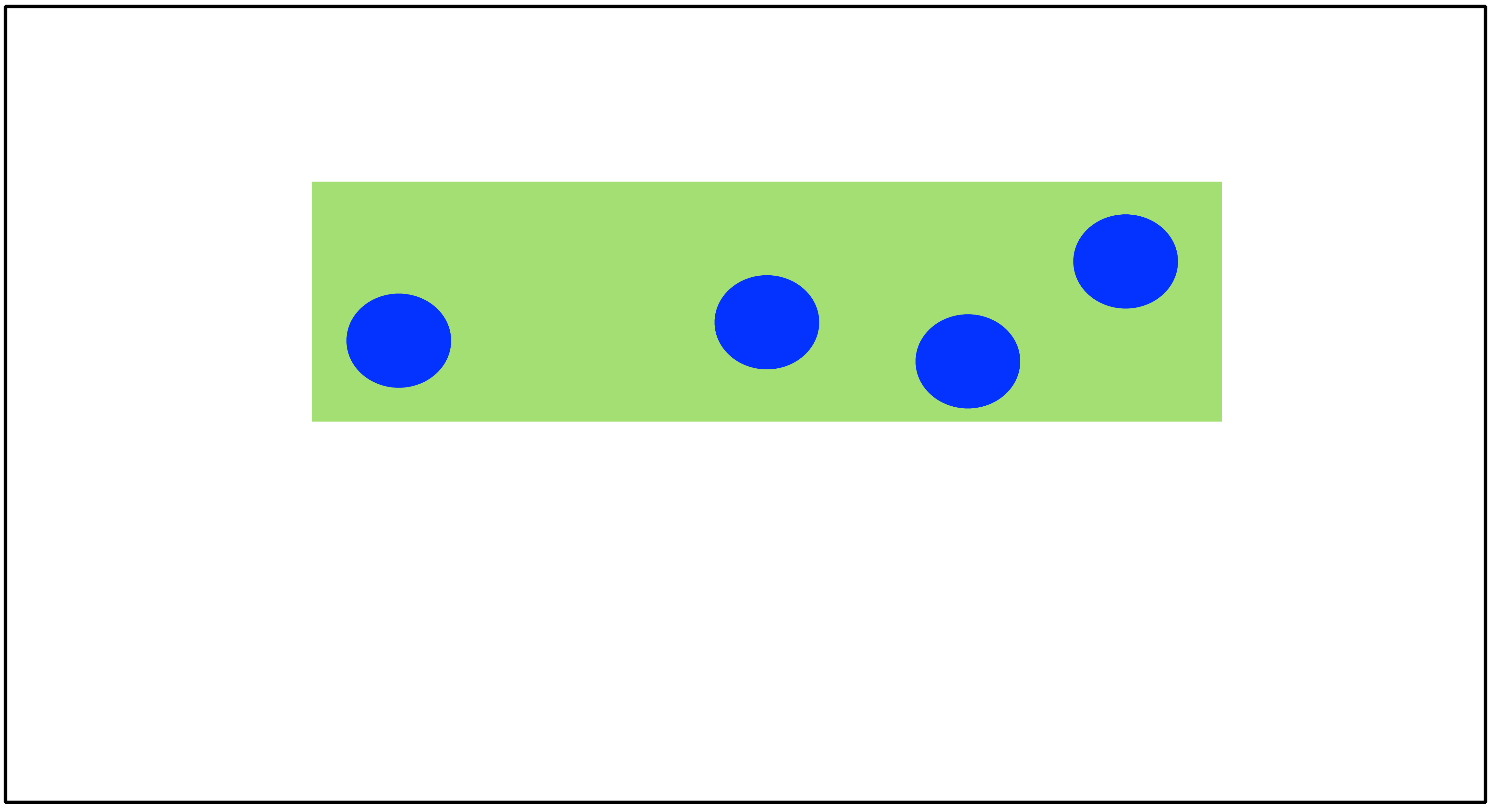
Outlier detector: an example

Reject task result for task 4



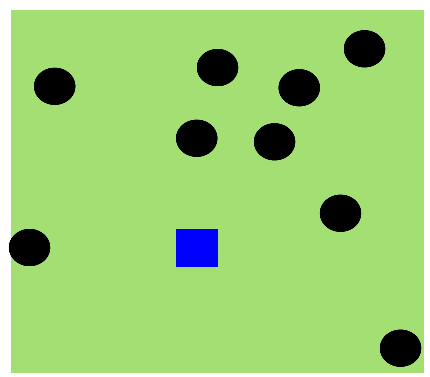
Outlier detector: an example

Reexecute task result for task 4

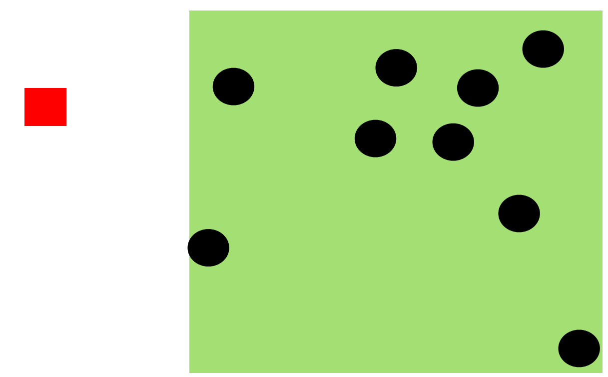


Outlier detection: possible outcomes

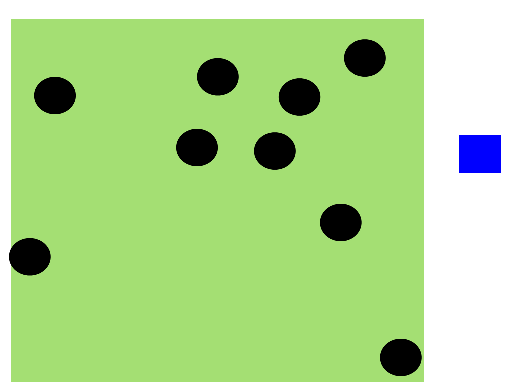
Correct result accepted



Error rejected

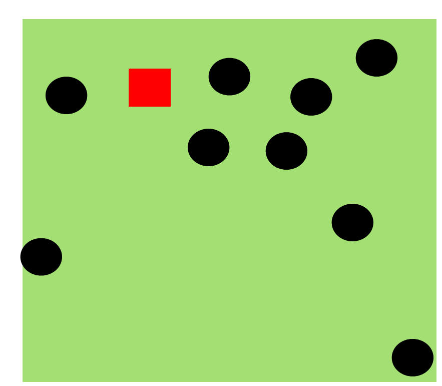


Correct result rejected



Incurs overhead

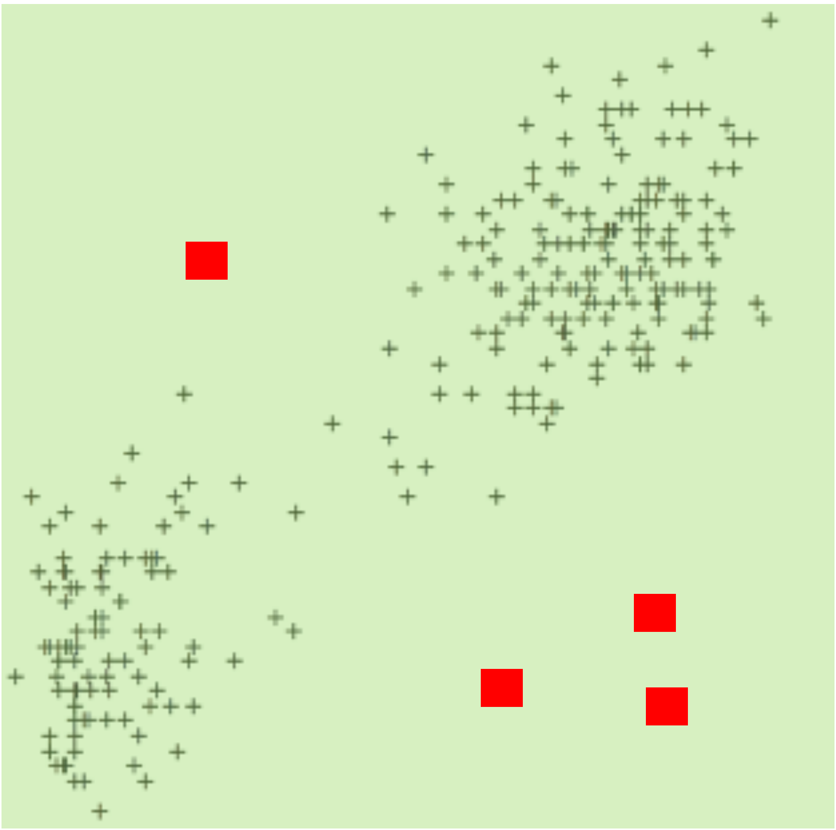
Error accepted



Error integrated into main computation

The basic outlier detector catches obvious task errors outside the result envelope

But, it **cannot** catch errors **between** the modes of a *multimodal result distribution*



Multi-region outlier detection

- **Multiple regions**
  - **N**: maximum number
  - **R**: set of hyperrectangle regions.
- **Algorithm**

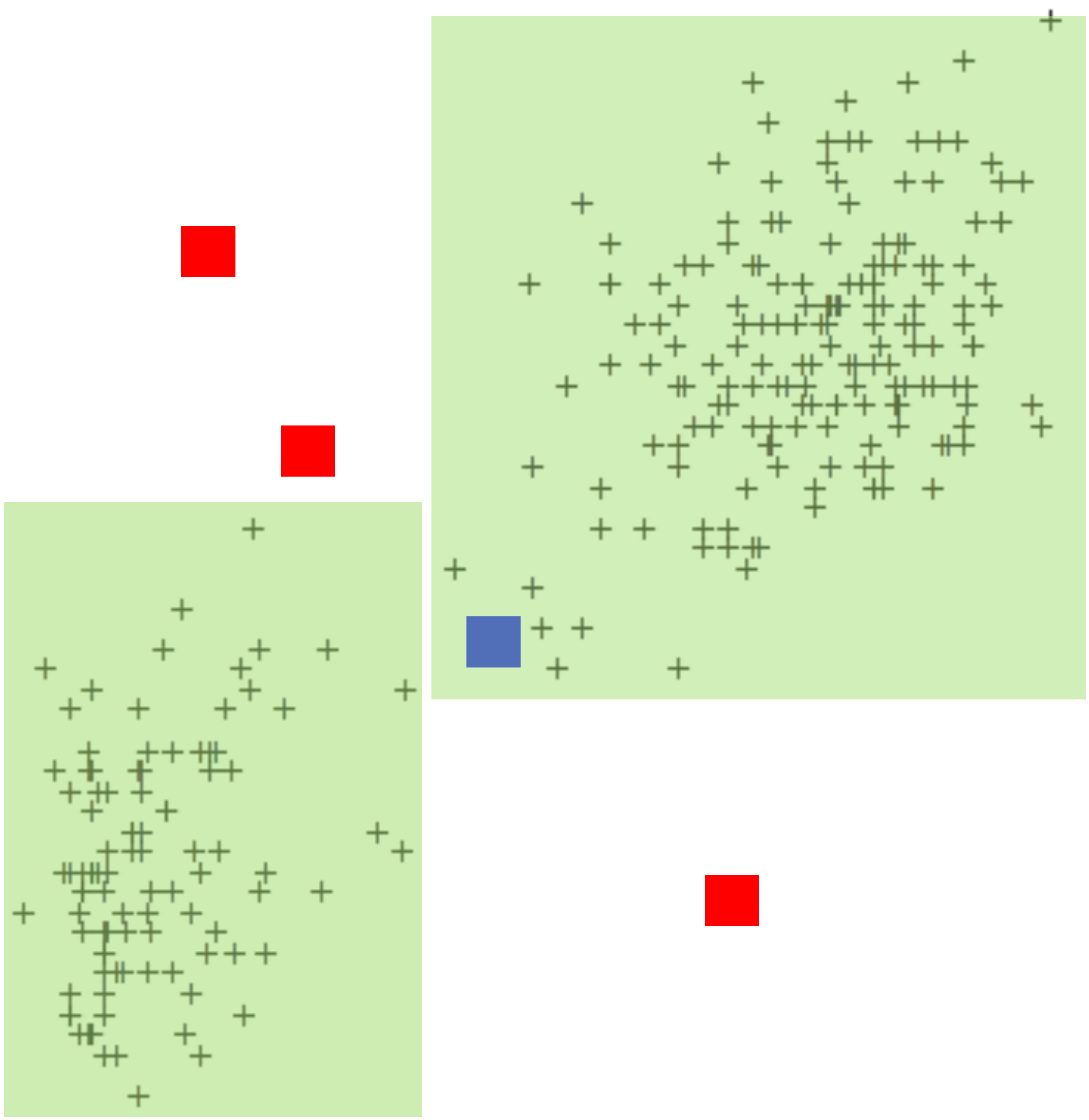
  Given result tuple x:

  If exists r in R s.t. x in r:

  Accept

  Otherwise

  Reject



Training the outlier detector

<u>Algorithm:</u> given rejected, reexecuted result x’

If dne r in R s.t. x’ in r:

Create region r’ where x’ in r’

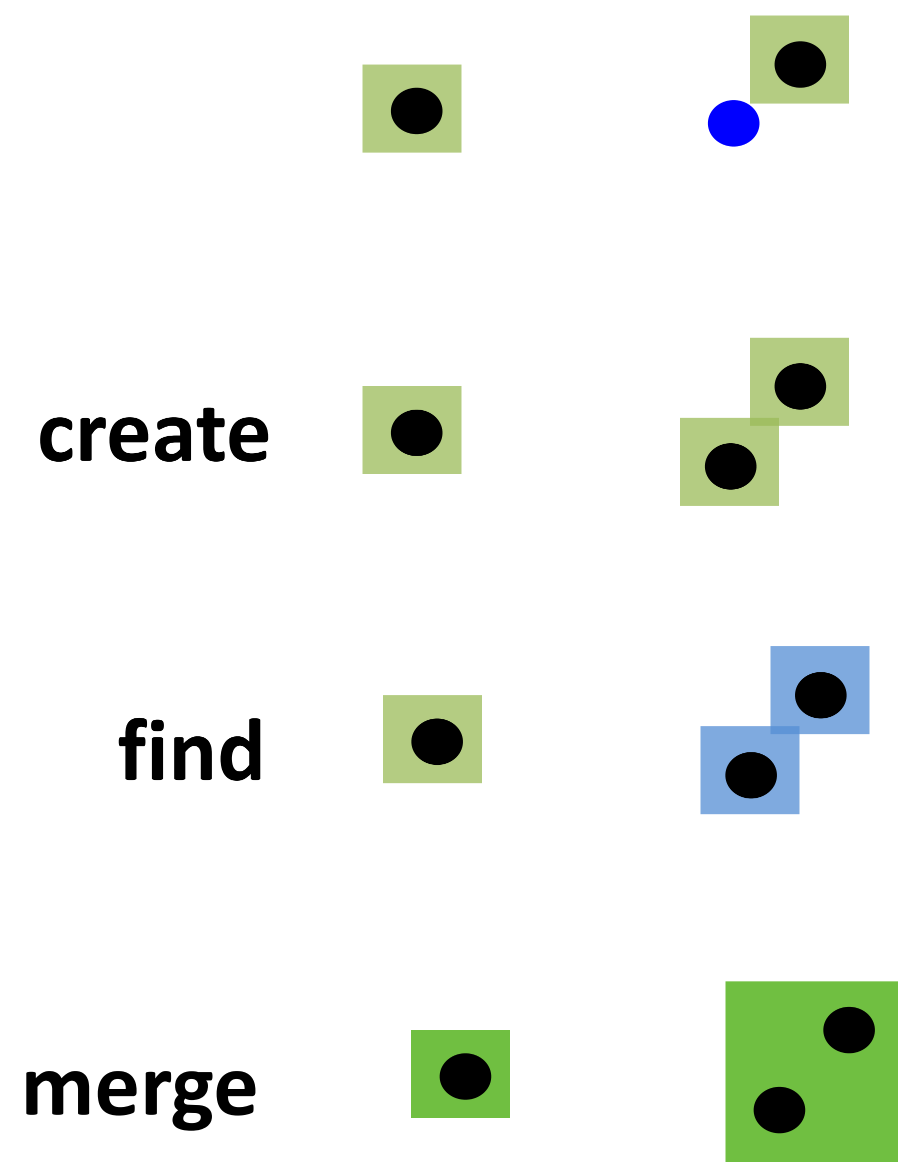
Add r’ to R

If |R| > N

Find two close regions r1,r2

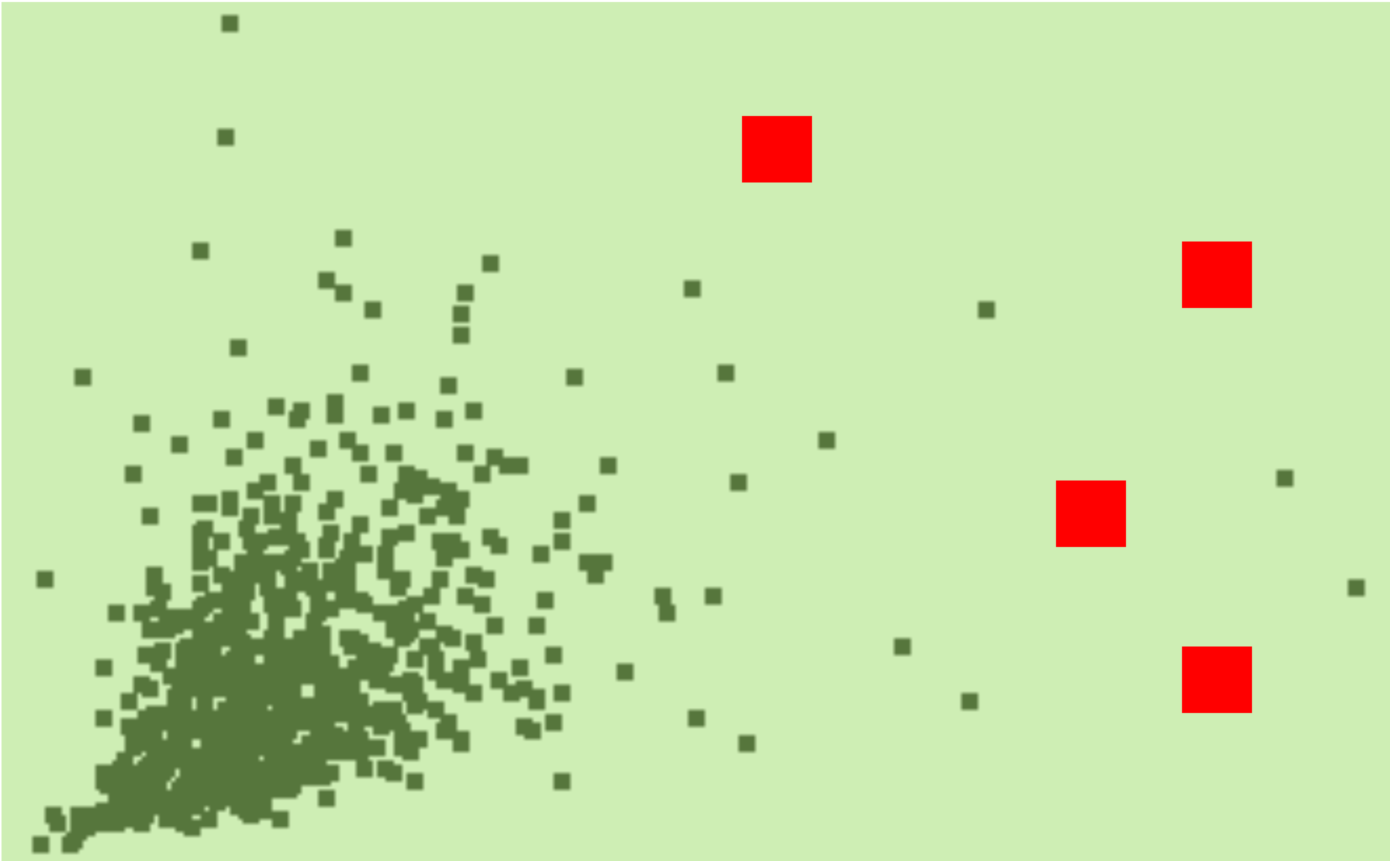
Merge r1 and r2

<u>Training on Reexecuted Result</u>
(N=2)



The multi-region outlier detector catches errors multimodal distributions

But, it **does not** catch errors in distributions that are **sparse** or **dynamic**



Adaptive Outlier Detection

- **Adaptive Outlier Detector**
  - **Vt**: target rejection rate
  - **Va**: actual rejection rate
  - **C**: control system
  - **COM(r)**: center of mass of region r
  - **Unlearn if we can reject more tasks**

- **Contract Algorithm**

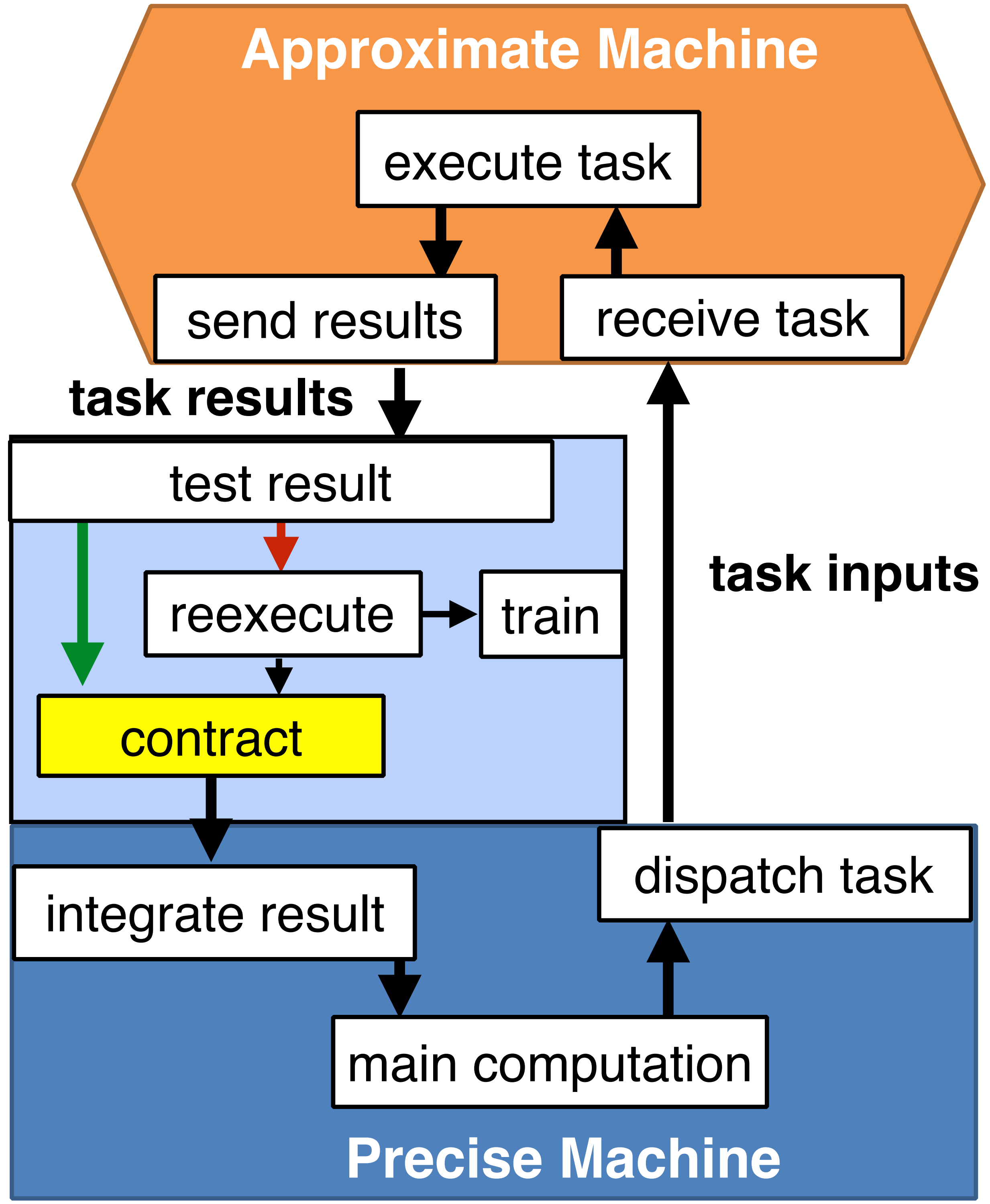
  Update COM(r) where x in r, r in R

  Update Va, C

  If Va < Vt:

    get factor f from C

    Contract all r in R by f



Adaptive Outlier Detection: *PID Control System*

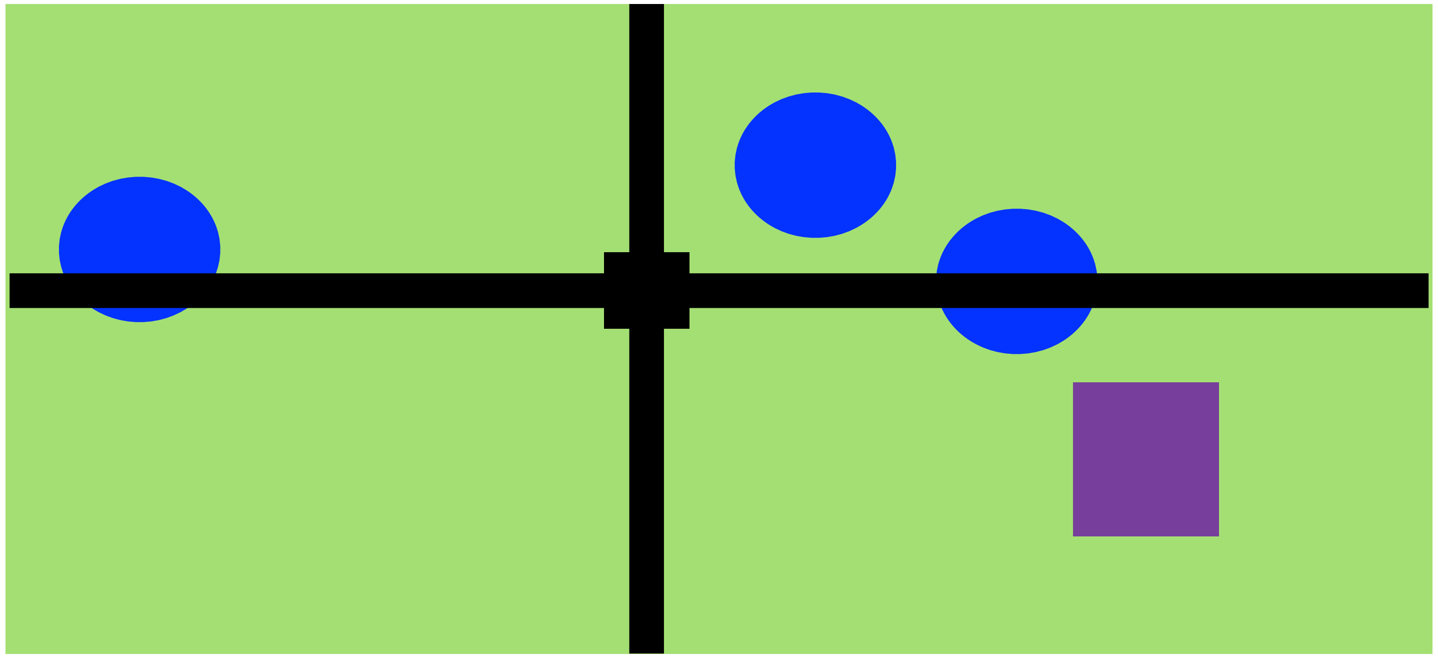
$$K_t \bullet e + K_d \bullet e' + K_i \bullet \int e$$

- ***Proportional-integral-derivative (PID)*** control
- **Error "value" (e)**: difference between measured (m), desired (d) value
  - <u>*measured value:*</u> **Va**, actual reexecution rate
  - <u>*desired value*</u>: **Vt**, target reexecution rate

Adaptive outlier detector: an example

Pass <u>result</u> from rejected and reexecuted task into contraction routine

C(Va,Vt) = 0.23



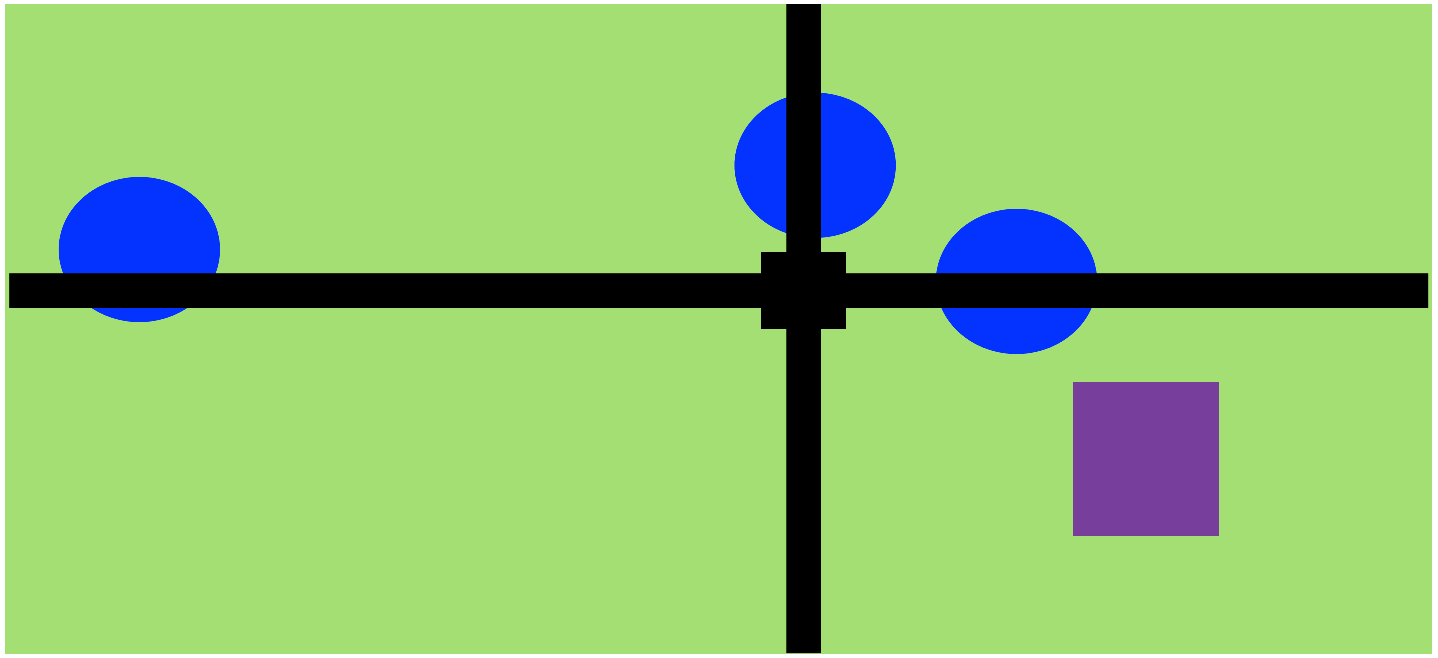
Va: 3% tasks rejected

Vt: 7% tasks rejected

Adaptive outlier detector: an example

Update center of mass of region

C(Va,Vt) = 0.23



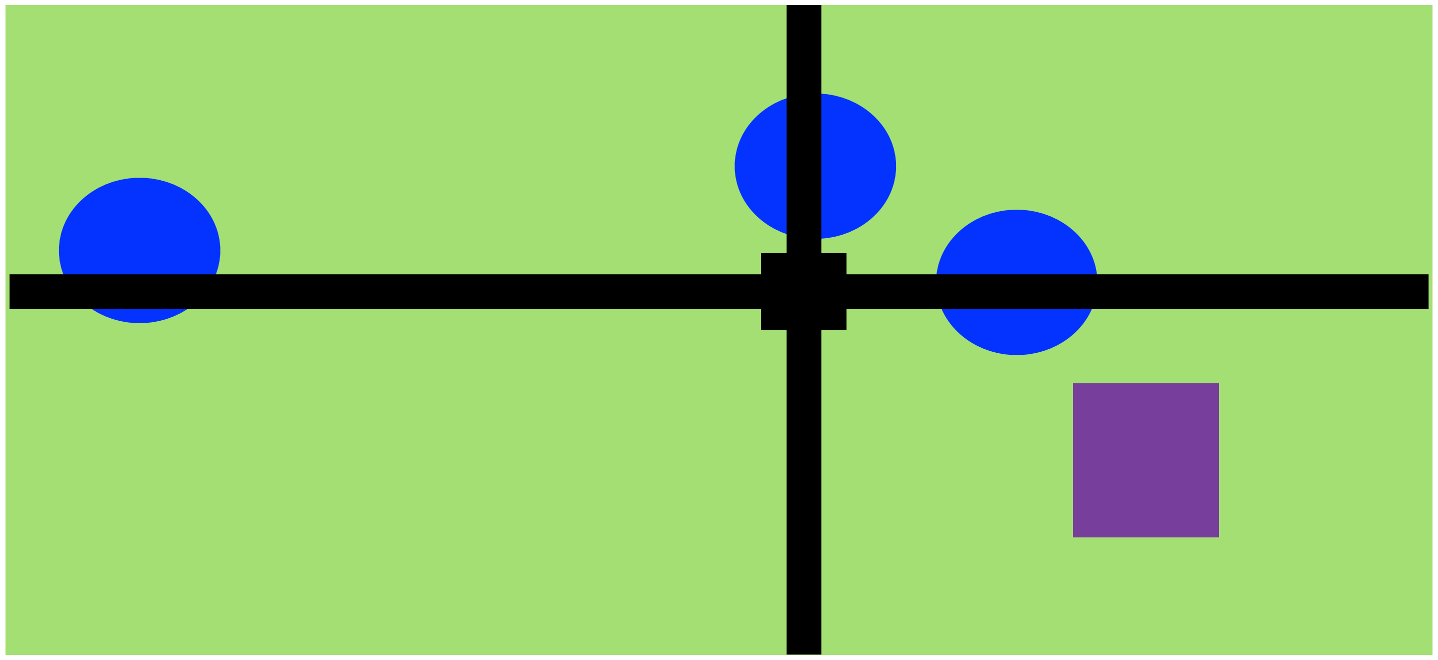
Va: 3% tasks rejected

Vt: 7% tasks rejected

Adaptive outlier detector: an example

Update *Va* to reflect rejected task. update *C*.

C(Va,Vt) = 0.19



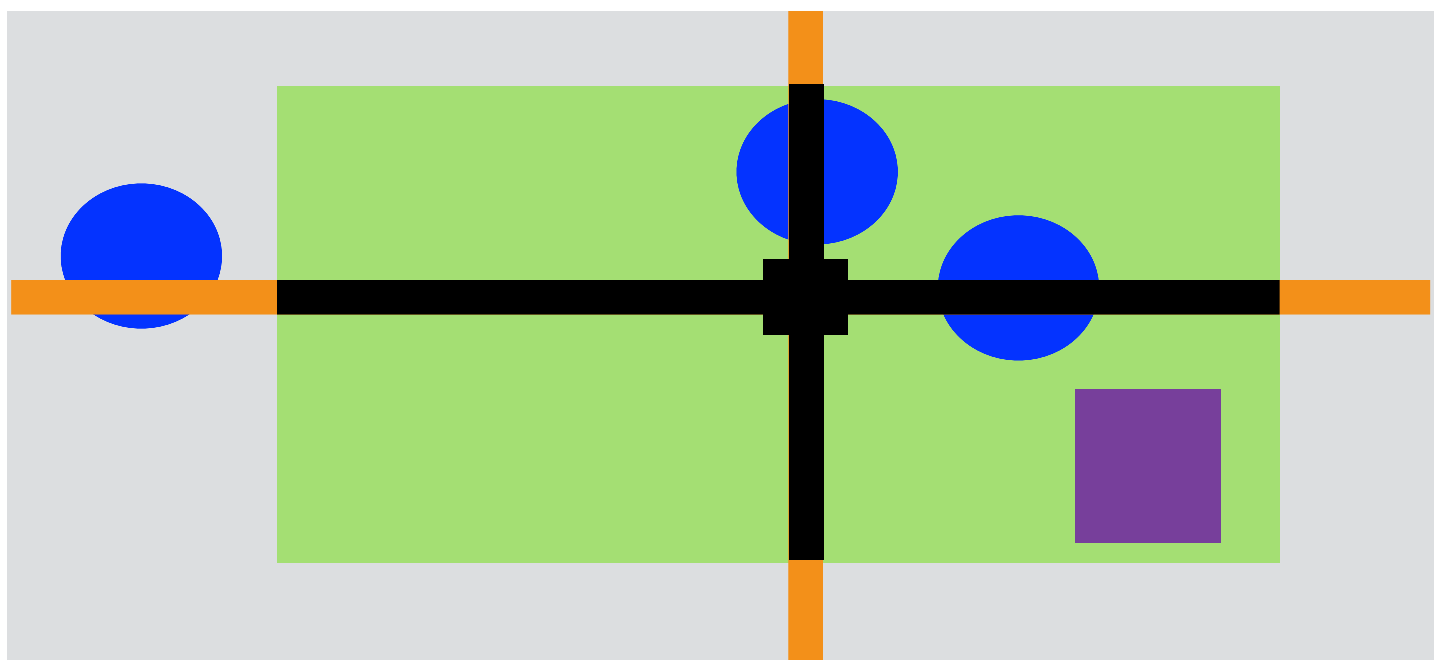
Va: 3.8% tasks rejected

Vt: 7% tasks rejected

Adaptive outlier detector: an example

***Va < Vt:* shrink the region by 19% about center of mass**

C(Va,Vt) = 0.19



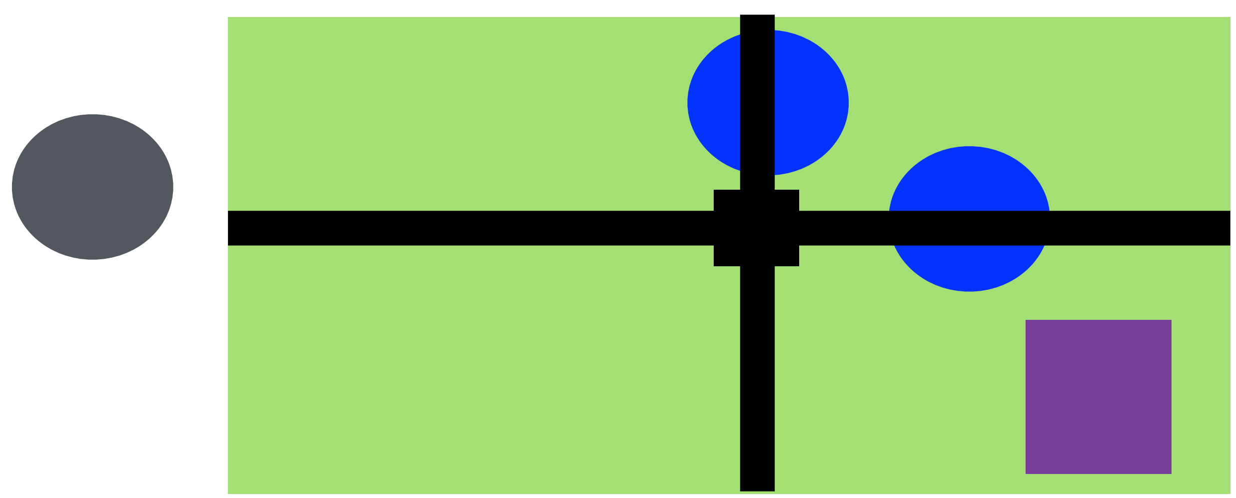
Va: 3.8% tasks rejected

Vt: 7% tasks rejected

Adaptive outlier detector: an example

Region successfully contracted

C(Va,Vt) = 0.19



Va: 3.8% tasks rejected

Vt: 7% tasks rejected

Topaz mitigates **crashes**

Language & computational model

Topaz corrects **unacceptable** task results

Outlier detection

Optimization 1: Stable Data

- **Stable Data**
  - data that is unchanged for all tasks in taskset
- **<u>Optimization</u>**: selectively send stable data
- **Reduce overhead if task contains large unchanging inputs**

```
// computes the weights for each valid pose.
taskset   calcweights(i=0; i<particles.size(); i+=1){
  compute in (
    float tpart[P_SIZE] =  (float*) particles[i],
    const float tmodel[M_SIZE] = (float*) mdl_prim,
    const char timg[I_SIZE] = (char *) img_prim,
    const int nCams = mModel->NCameras(),
    const int nBits = mModel->getBytesPerPixel(),
    const int width = mModel->getWidth(),
    const int height =mModel->getHeight()
  ) out (float tweight) {
      tweight = CalcWeight(tpart,
          tmodel, timg, nCams, width, height, nBits);
  }
```

`const` stable data annotation for inputs

Optimization 2: Abstract Output Vector (AOV)

- **Abstract output vector (AOV)**
  - Programmer defined result tuple abstraction.
- **<u>Optimization</u>**: perform detection on smaller AOV.
- **<u>Aside</u>**: handle input dependence using AOV
- **Reduces outlier detector overhead if AOV smaller than result tuple**

```
taskset   name(int i = l; i < u; i++) {
  compute in  (d1 x1 = e1, ..., dn xn = en)
              out  (o1 y1, ..., oj yj) {
    <task body>
  }
  transform out  (v1, ..., vk) {
    <output abstraction>
  }
  combine { <combine body> }
}
```

`transform` block with AOV outputs `v1..vk`.

`<output abstraction>` defines the transformation

Does Topaz perform well in **practice**?

Experimental Setup

- **Hardware Model**
- No-refresh DRAM
  Protections removed
  - Bit, time dependent errors
- Dual-voltage L1, L2 caches
  Aggressive conditions
  - Per-read / per-write errors
- **Benefit**: saves energy

- **Benchmarks**
- Barnes: planet simulation
- Bodytrack: machine vision
- Water: water simulation
- Blackscholes: financial analysis
- Streamcluster: k-means clustering

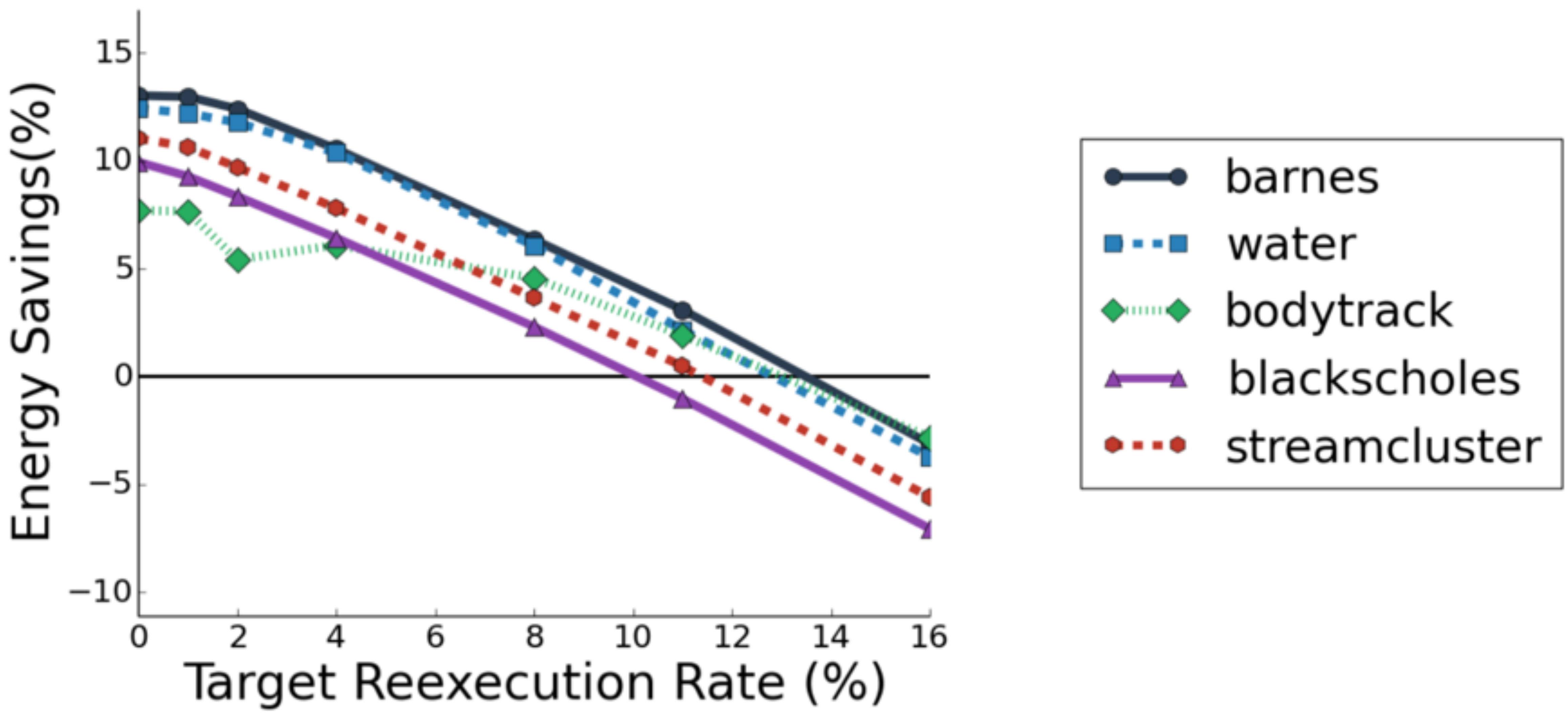
Question: What sorts of **energy savings** do we observe with this system

Energy savings

Benchmarks	Model	Baseline	Detect & Reexecute	Full Topaz
barnes	basic	17.47%	14.77%	13.02%
blackscholes	basic	16.20%	14.62%	9.94%
bodytrack	basic	12.70%	8.60%	7.69%
streamcluster	basic	16.87%	15.62%	11.03%
water	basic	18.41%	15.12%	12.43%
barnes	ddep	17.47%	14.76%	13.02%
blackscholes	ddep	16.02%	14.41%	9.70%
bodytrack	ddep	12.88%	6.51%	5.02%
streamcluster	ddep	16.89%	15.58%	11.03%
water	ddep	18.41%	15.37%	12.82%

Table 4: Energy Savings, basic and ddep Hardware Models

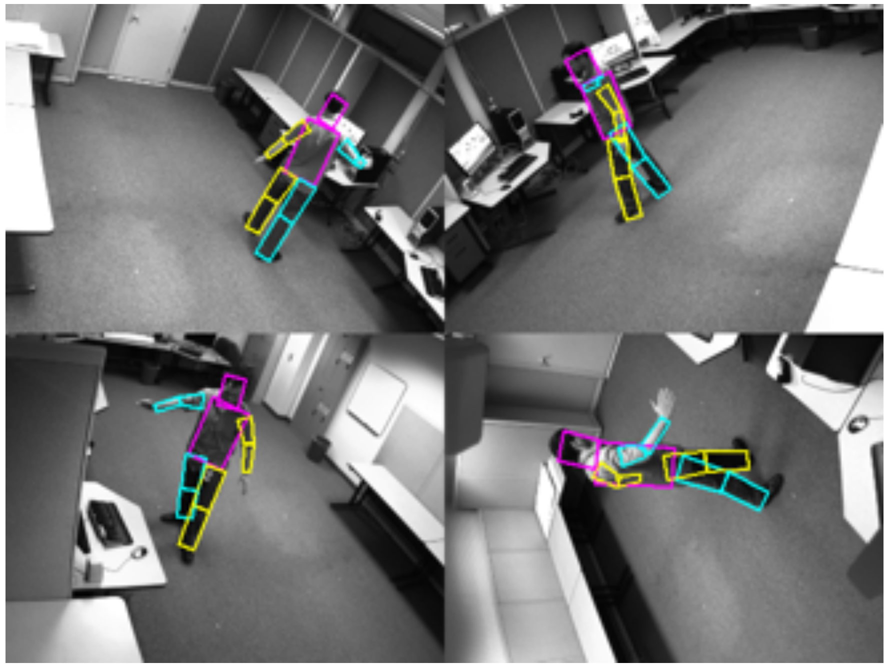
Energy savings

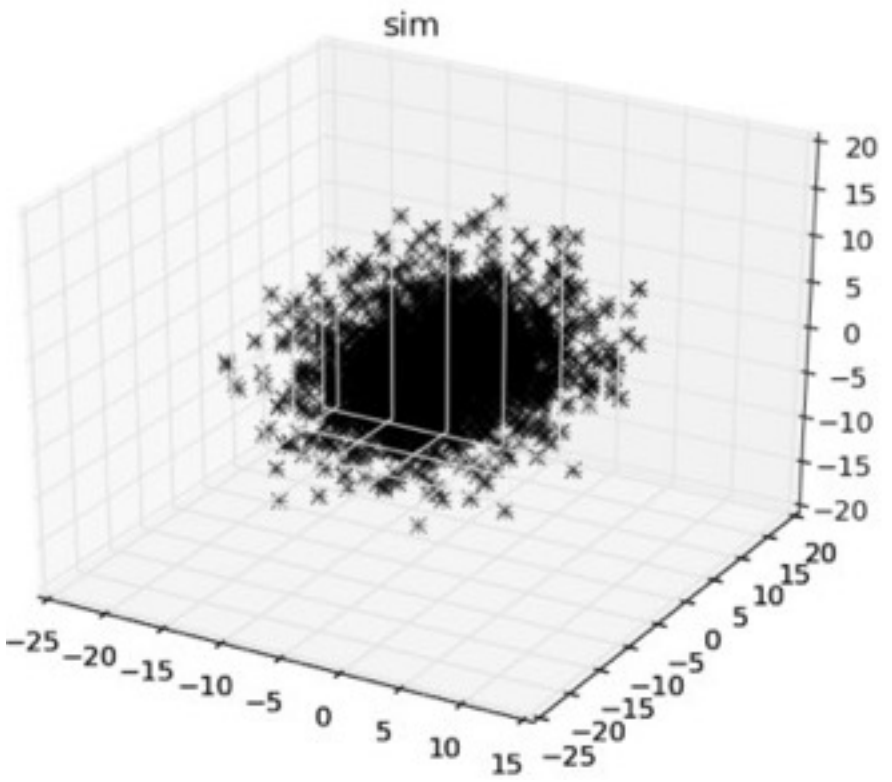


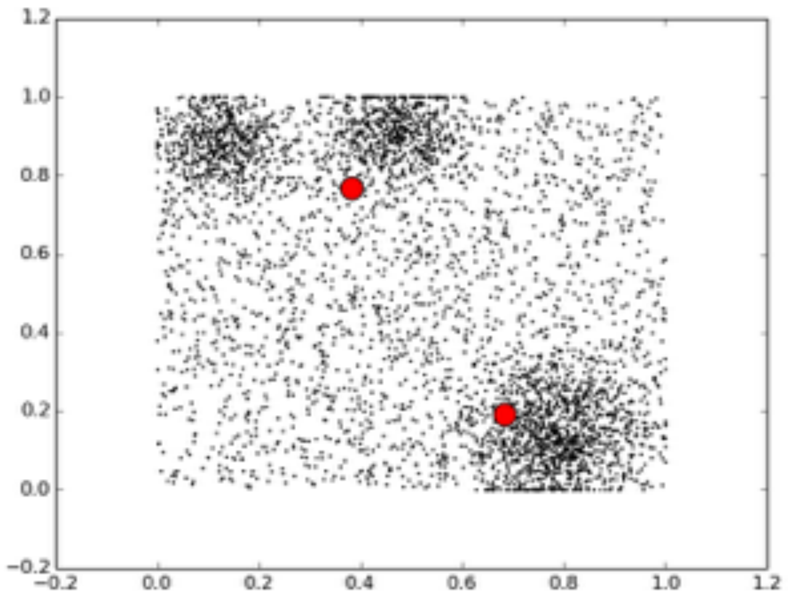
(d) Energy Savings

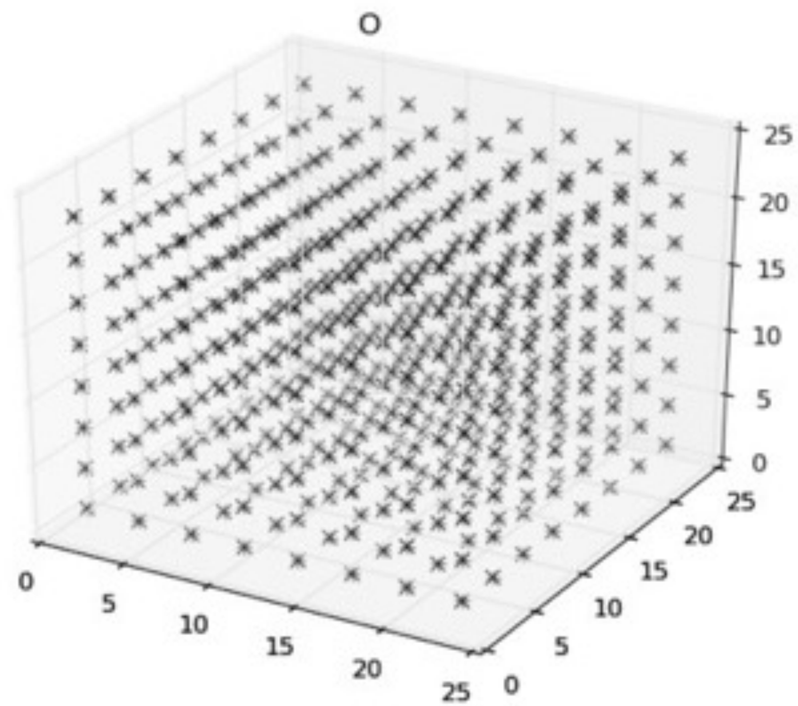
Question: Are the end-to-end results acceptable?

Expected Result

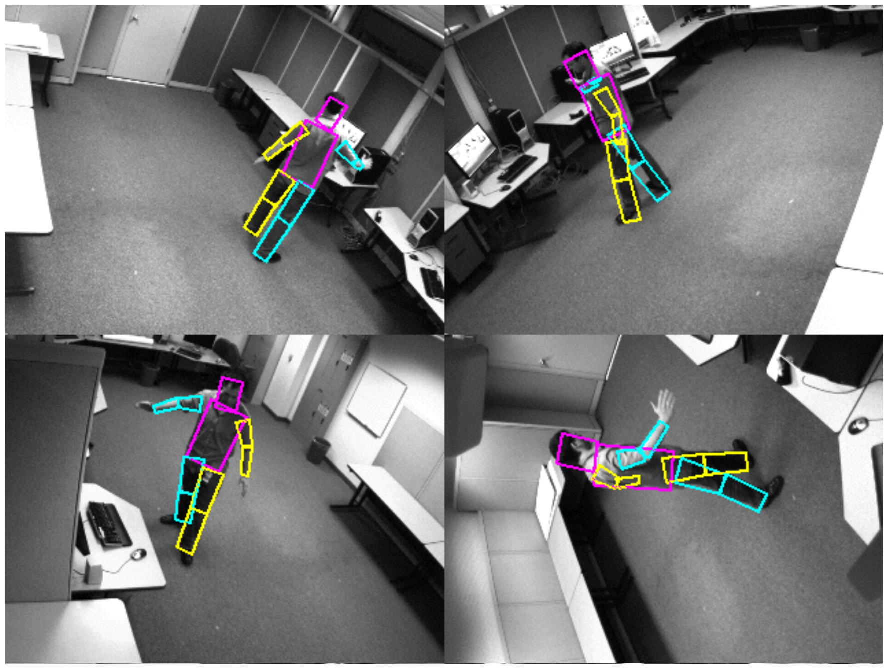


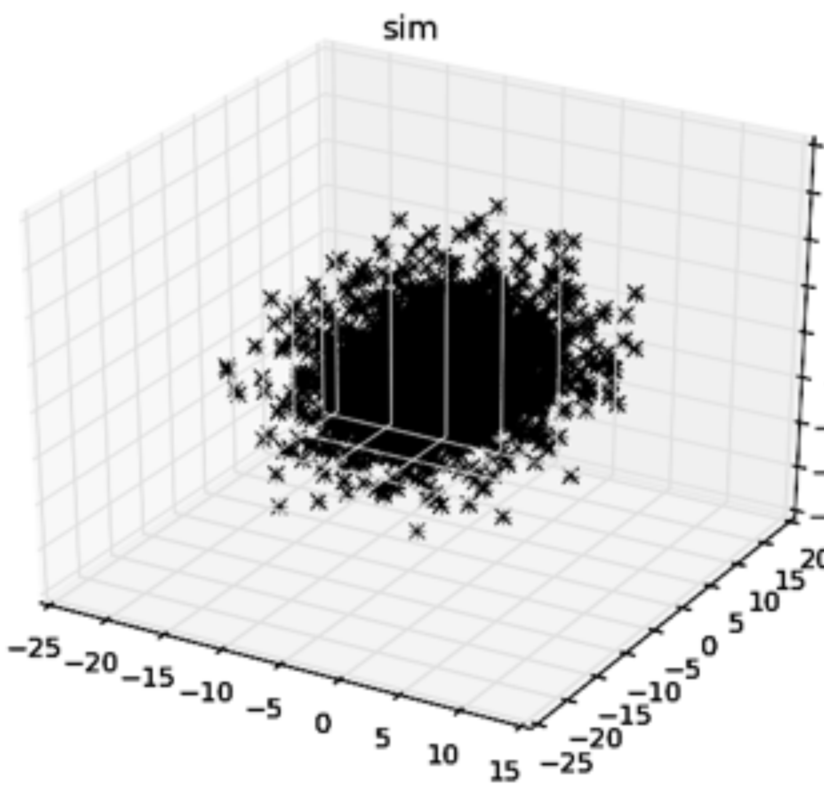


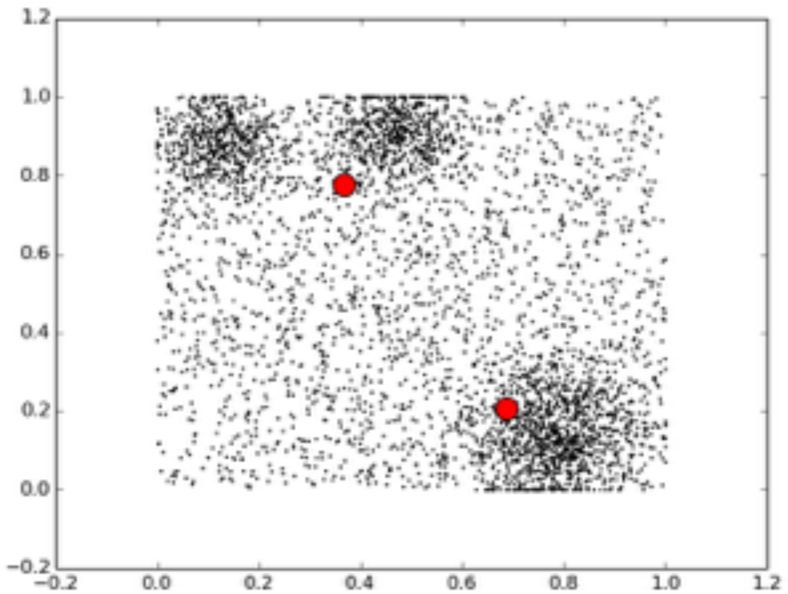


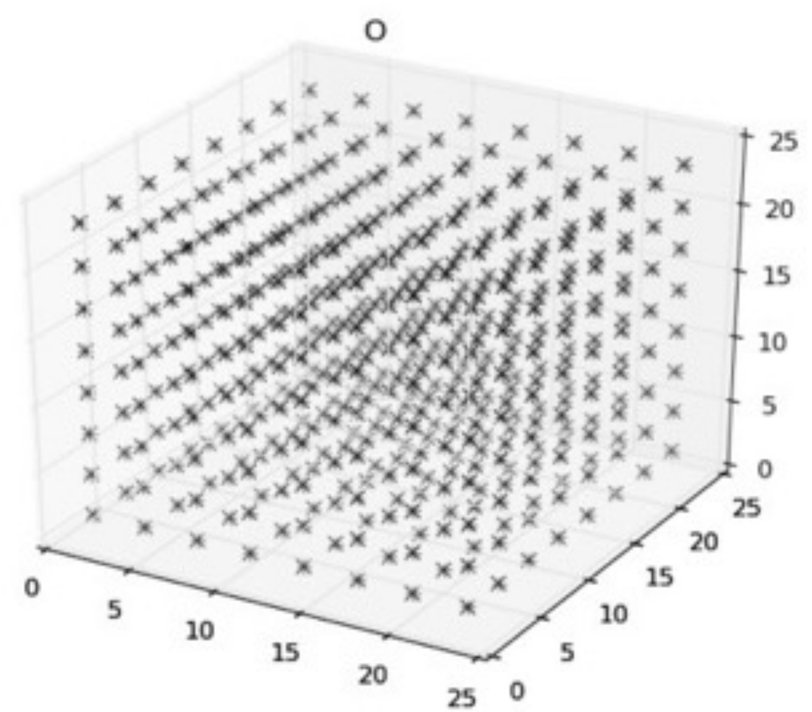


Approximate Result **with** Outlier Detection

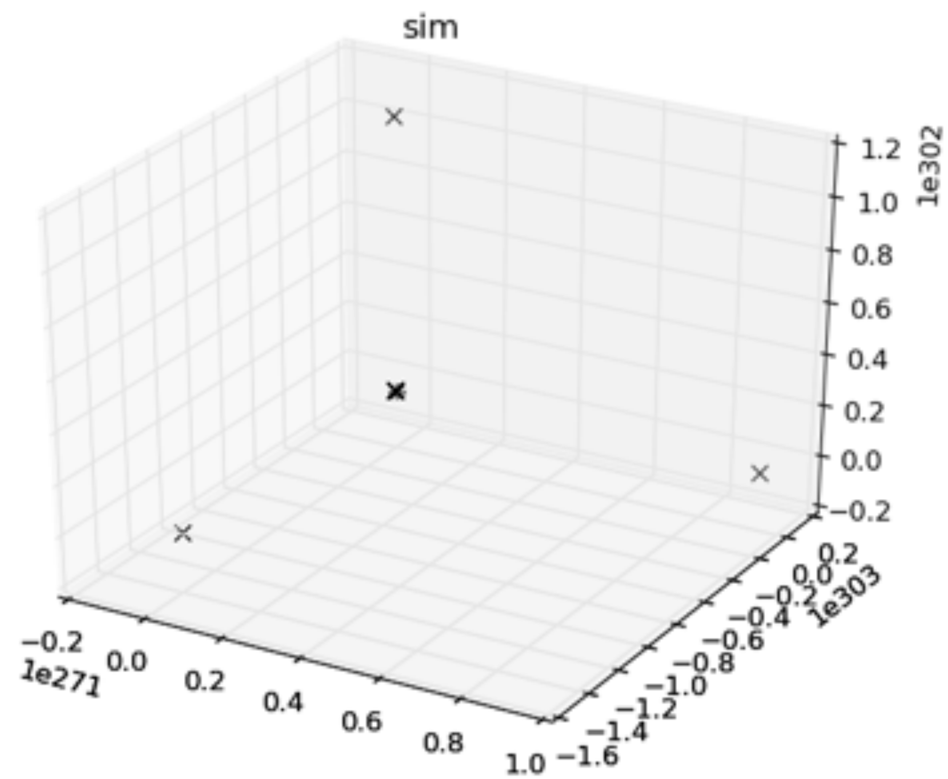


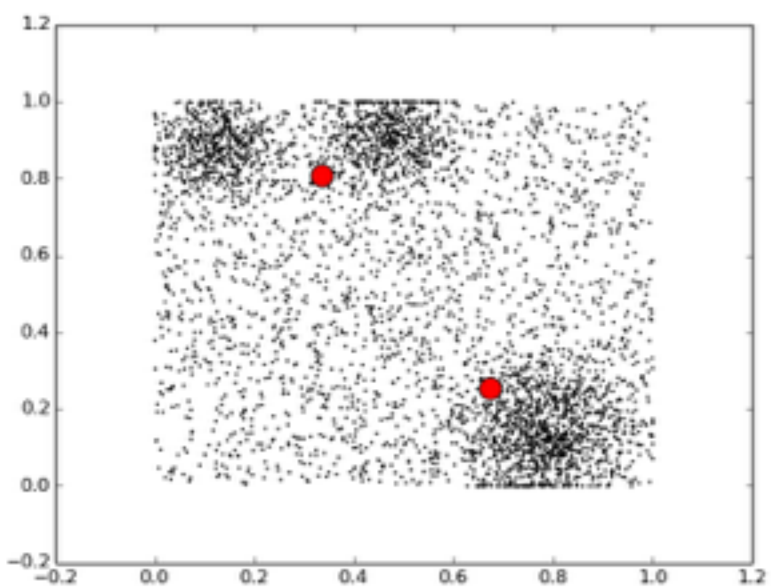


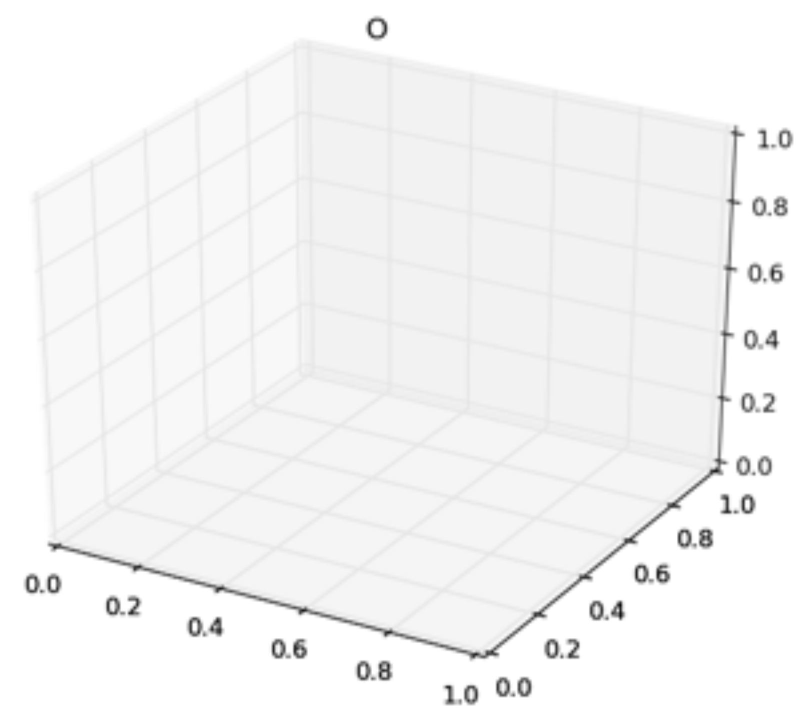




Approximate Result **without** Outlier Detection







Quantitative Analysis of Final Result Quality

Benchmark	Model	No Outlier Detector	Outlier Detector
barnes	basic	inf	0.158229%
blackscholes	basic	inf	0.135584%
bodytrack	basic	73.6327%	0.161024%
streamcluster	basic	0.6219	0.6344
water	basic	nan	0.000469%
barnes	ddep	inf	0.075927%
blackscholes	ddep	inf	0.025791%
bodytrack	ddep	73.6327%	0.317984%
streamcluster	ddep	0.6321	0.6344
water	ddep	nan	0.000383%

Table 2: End-to-End Output Quality

Question: Is the outlier detector adequately detecting outliers?

Quantitative Outlier Detector Efficacy

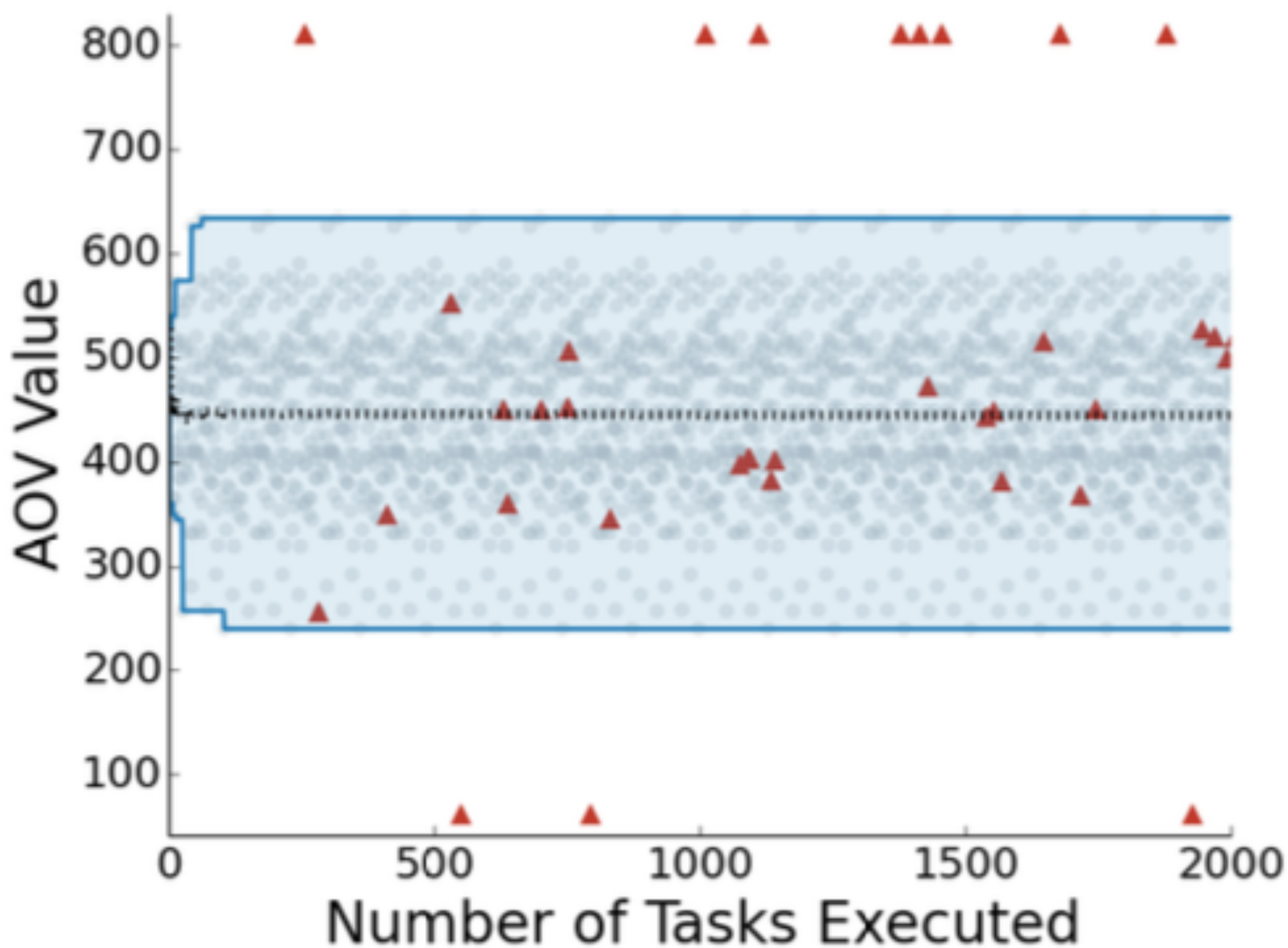
Benchmark	Hardware Model	Correct Accepted (%)	Correct Rejected (%)	Error Accepted (%)	Error Rejected (%)	Rejection Accuracy (%)	Errors Detected (%)
barnes	basic	94.48%	0.19%	2.94%	2.38%	92.58%	44.74%
bodytrack	basic	87.58%	0.16%	7.67%	4.58%	96.62%	37.39%
water-interf	basic	95.30%	0.32%	1.71%	2.67%	89.37%	60.96%
water-poteng	basic	99.51%	0.26%	0.02%	0.20%	43.59%	89.47%
blackscholes	basic	98.57%	0.04%	1.06%	0.33%	90.00%	24.06%
streamcluster	basic	98.34%	0.14%	0.37%	1.15%	89.15%	75.66%
barnes	ddep	94.22%	0.20%	3.11%	2.47%	92.59%	44.26%
bodytrack	ddep	77.34%	0.15%	16.04%	6.46%	97.67%	28.71%
water-interf	ddep	95.44%	0.33%	1.62%	2.61%	88.81%	61.71%
water-poteng	ddep	99.49%	0.26%	0.04%	0.21%	44.54%	85.48%
blackscholes	ddep	98.70%	0.04%	0.94%	0.33%	89.80%	25.88%
streamcluster	ddep	62.24%	0.11%	36.68%	0.98%	89.90%	2.59%

Table 3: Overall Outlier Detector Effectiveness

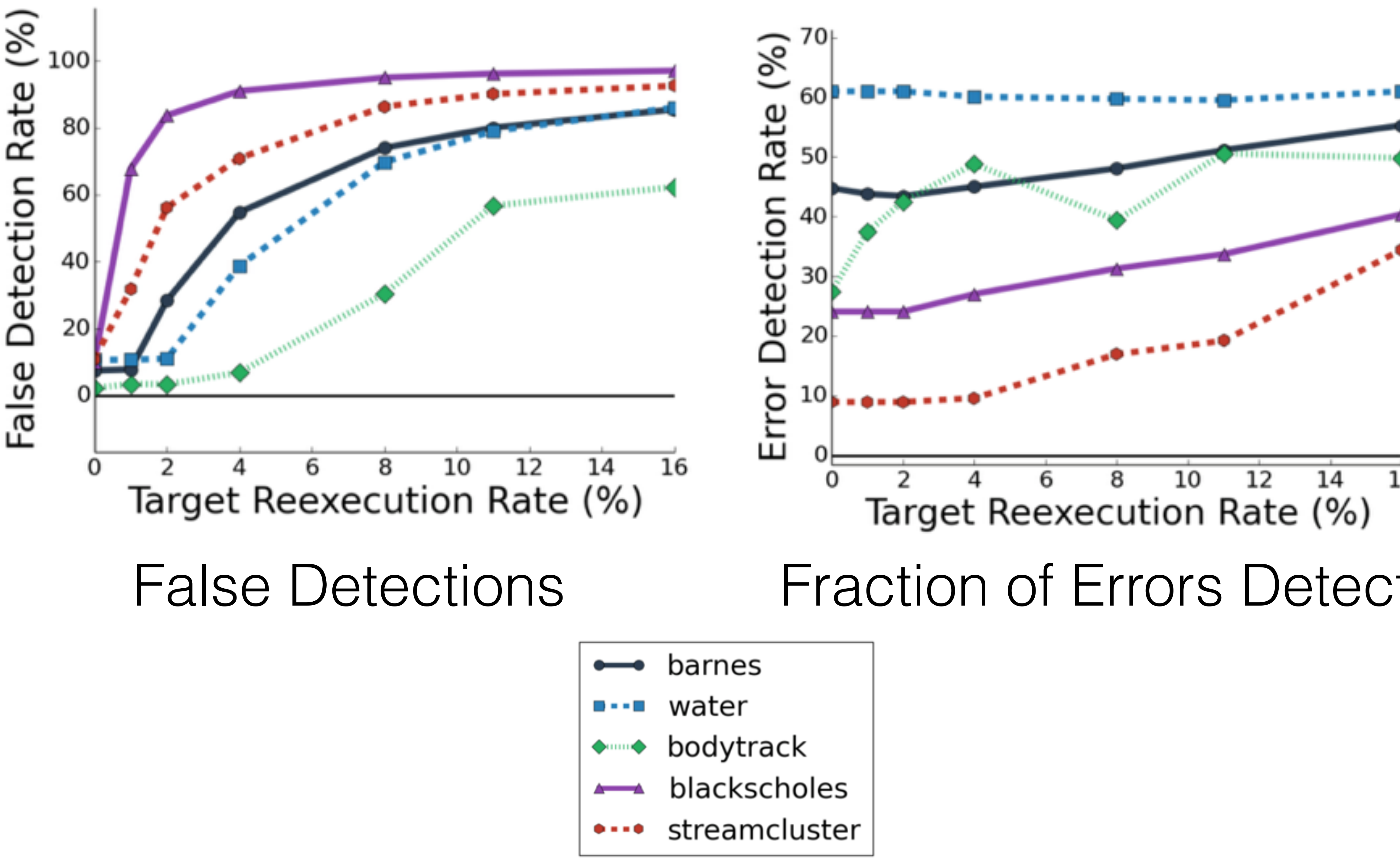
- **Some tasks are rejected:** 0.5-7% tasks rejected
- **Most rejections are errors:** 87%-98% rejections are errors
- **Some errors are undetected:** 2%-90% errors are rejected

Qualitative Error Characteristics and Detection

Blackscholes: 24% of errors detected



Qualitative Error Characteristics and Detection

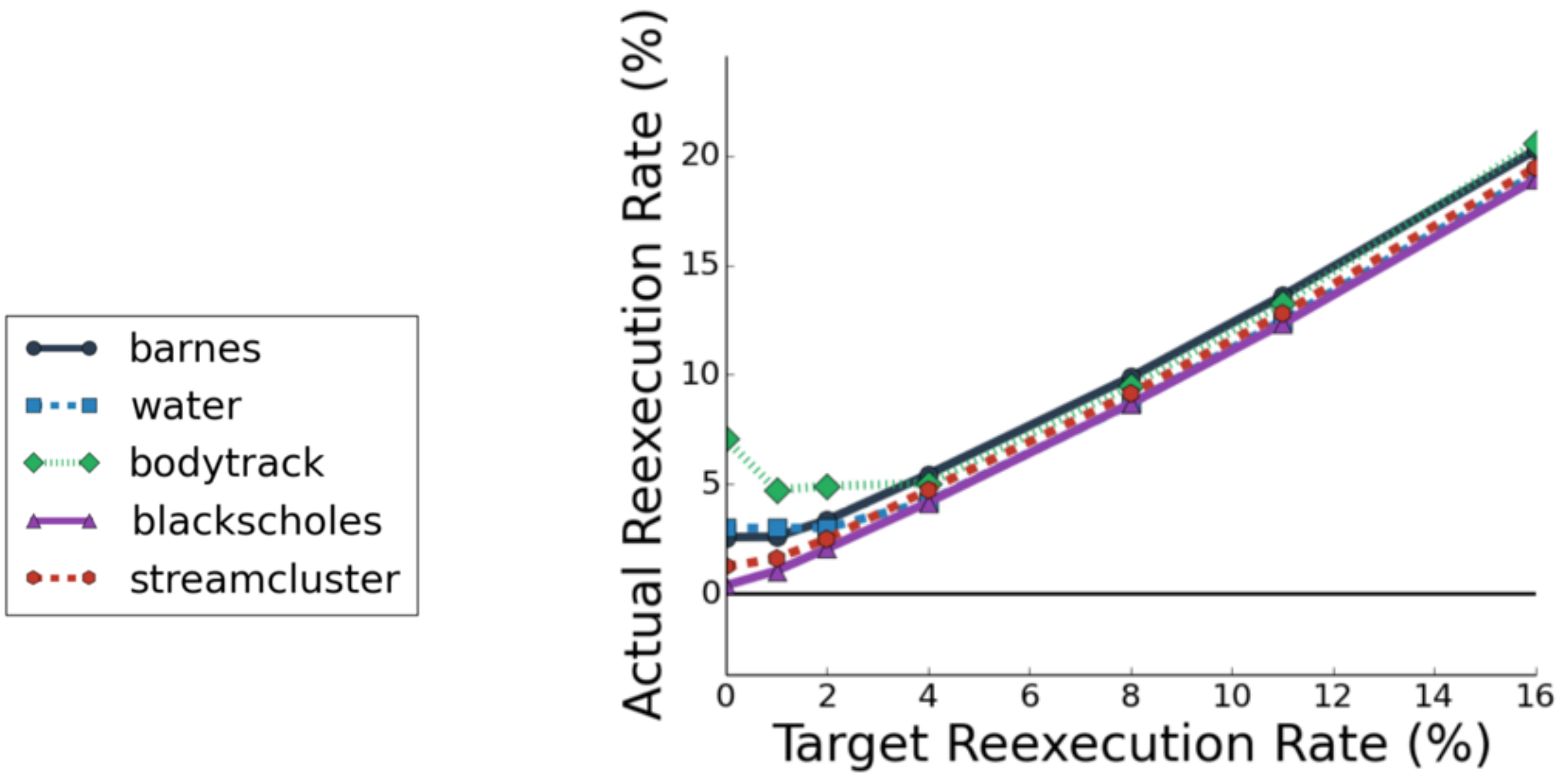


False Detections

Fraction of Errors Detected

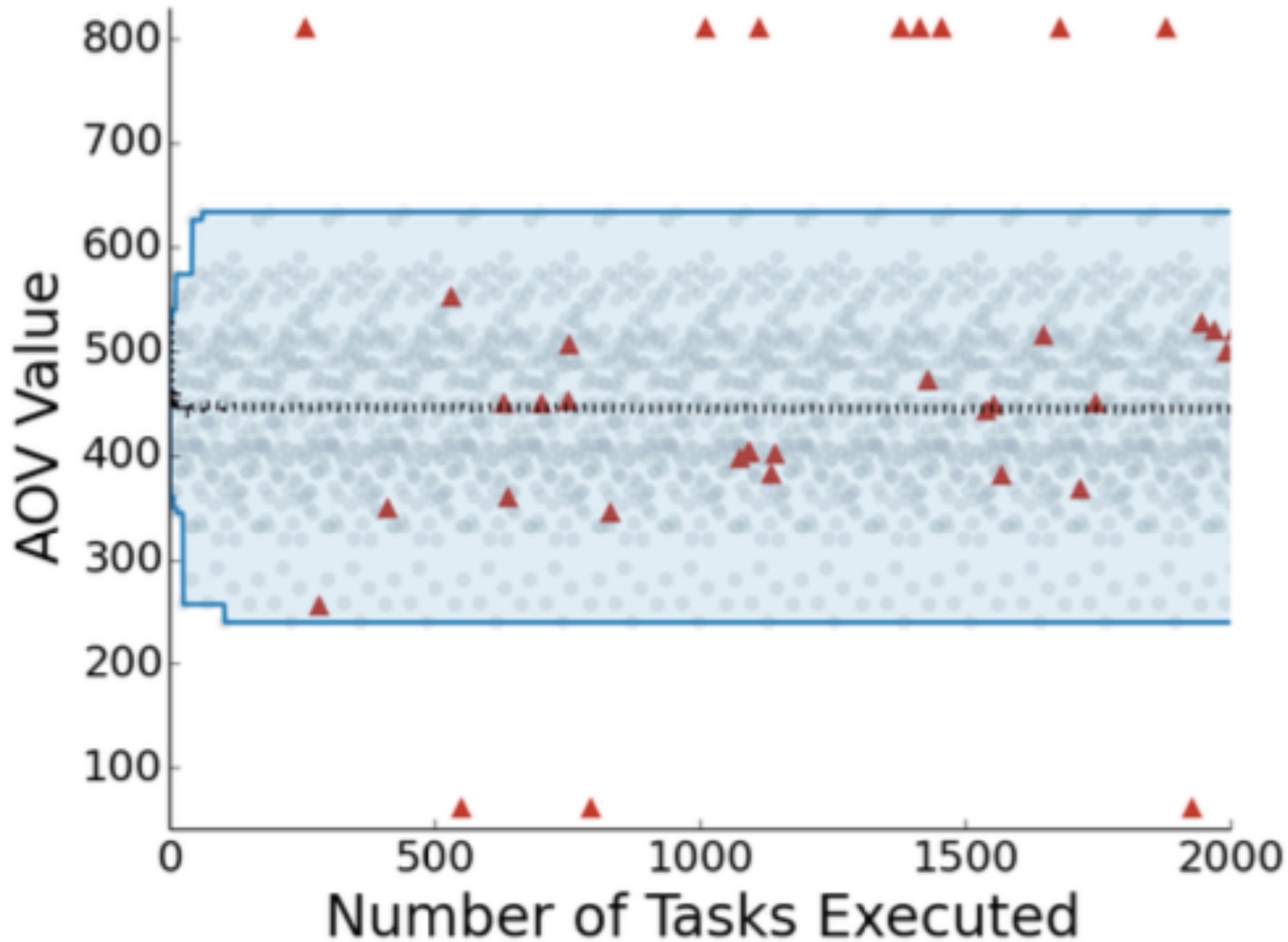
Question: Is the adaptive outlier detector behaving as expected?

Quantitative Efficacy of Adaptive Outlier Detector



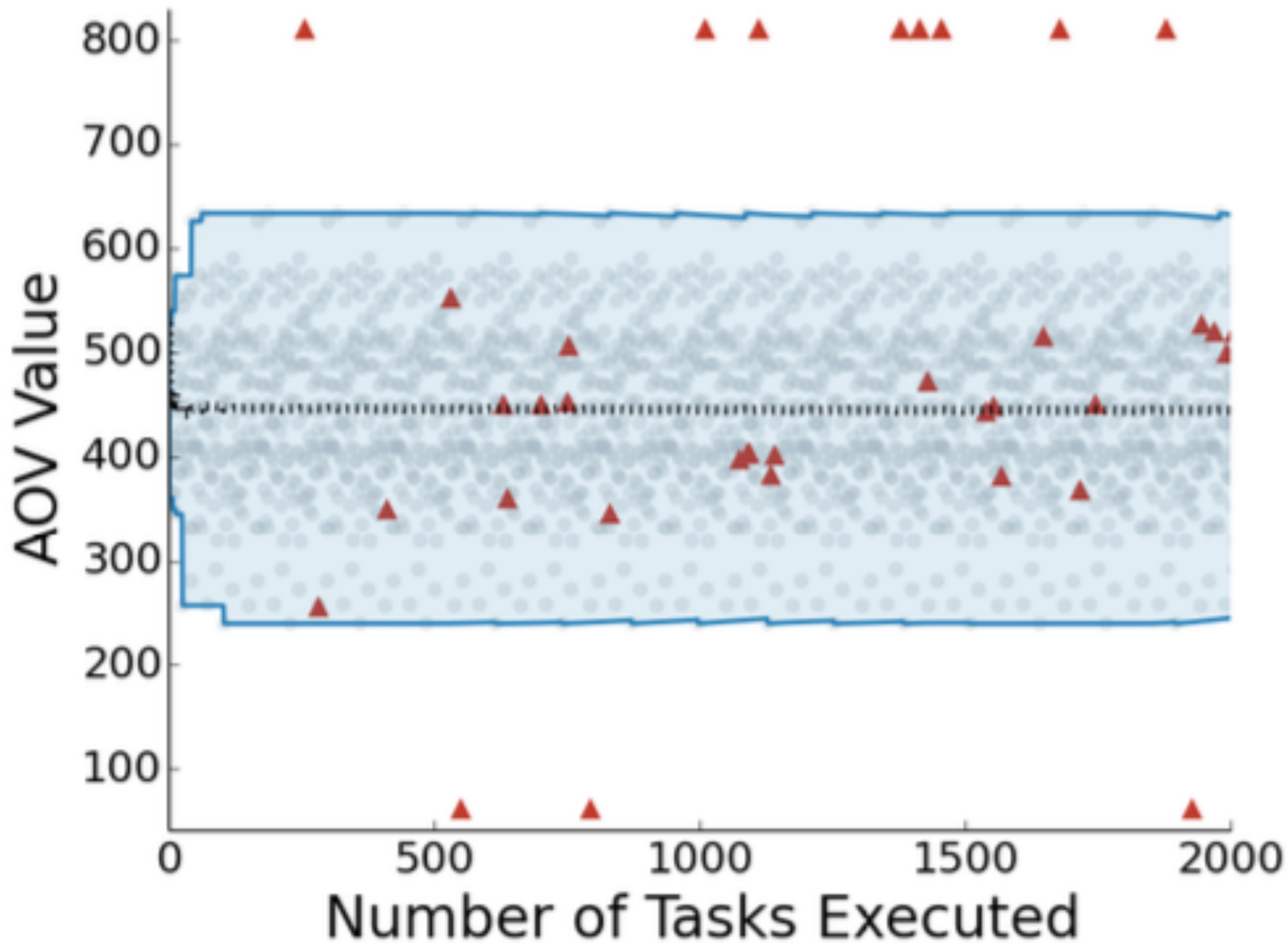
(a) Actual Reexecution Rate

Qualitative adaptive outlier detector efficacy



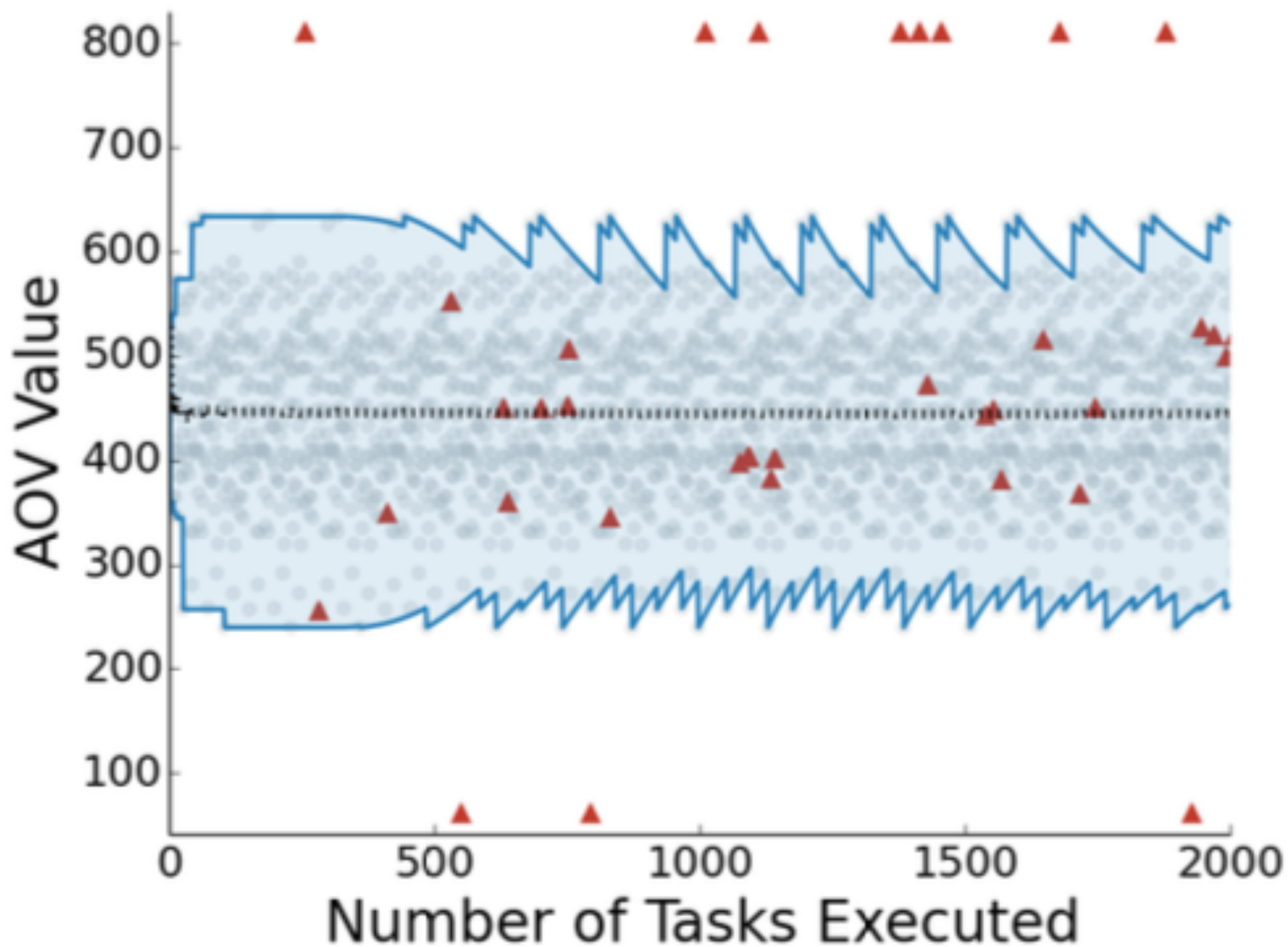
1% target reexecution rate

Qualitative adaptive outlier detector efficacy



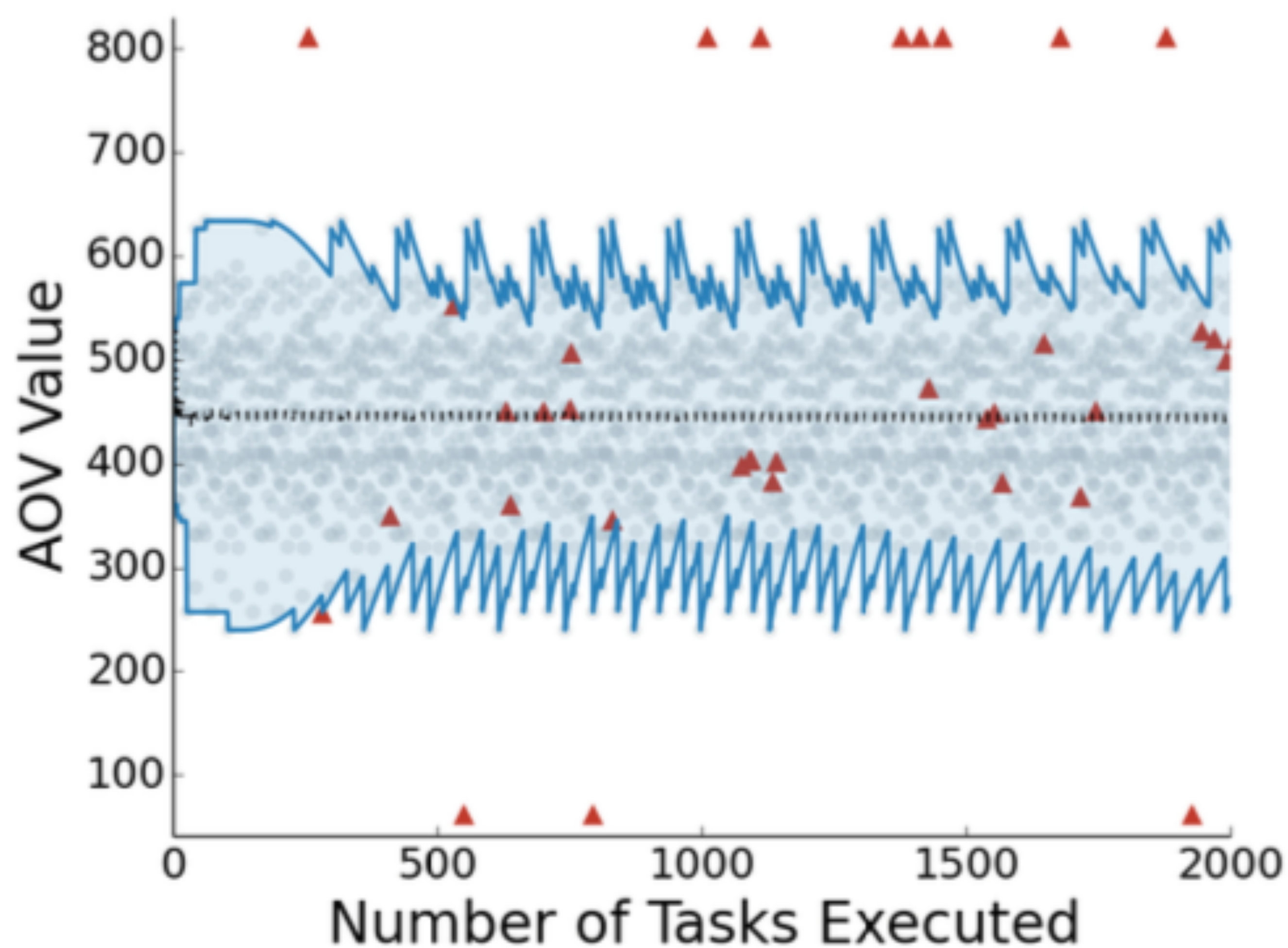
2% target reexecution rate

Qualitative adaptive outlier detector efficacy



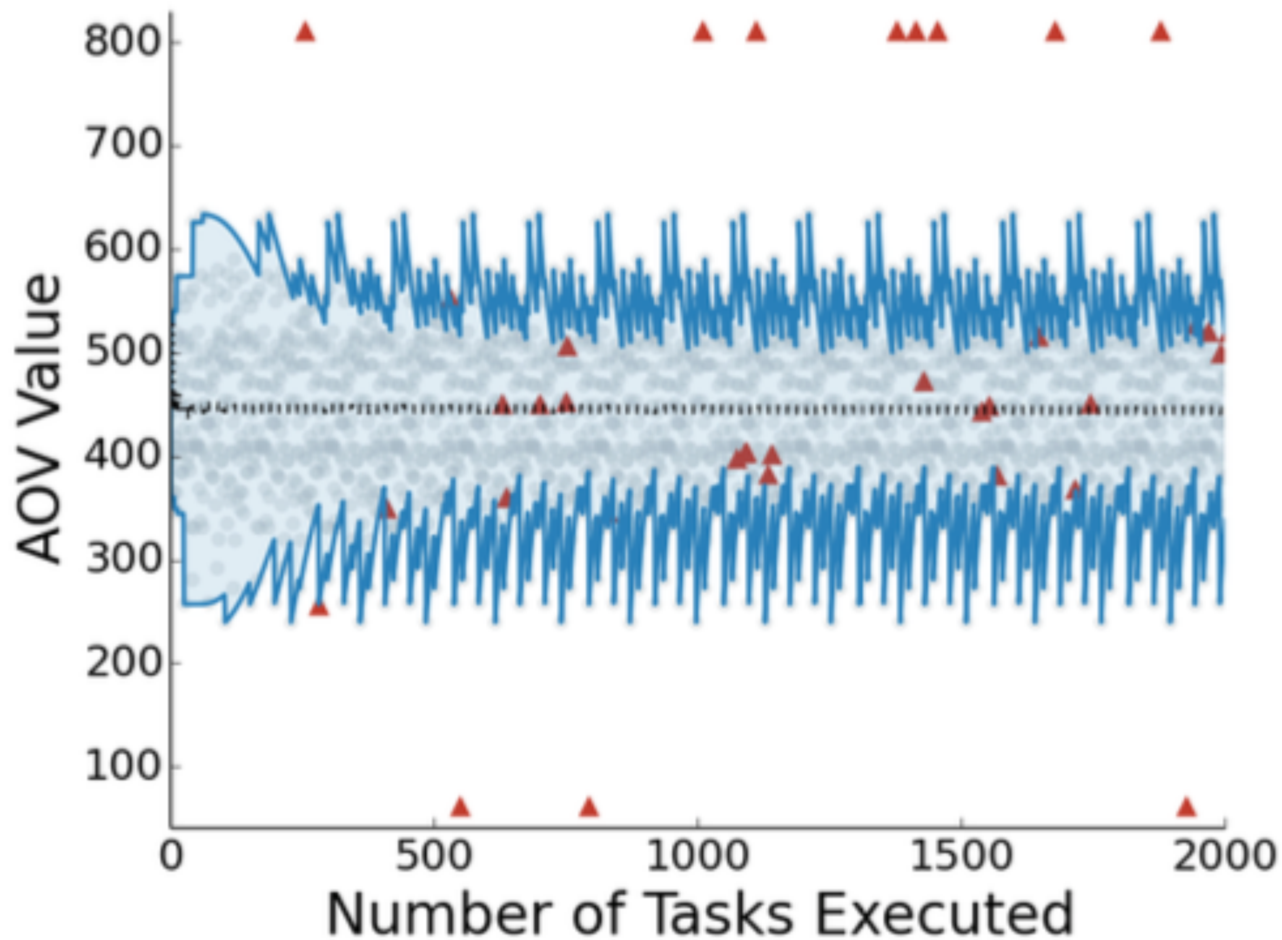
4% target reexecution rate

Qualitative adaptive outlier detector efficacy



8% target reexecution rate

Qualitative adaptive outlier detector efficacy



16% target reexecution rate

Conclusion

- ***Topaz*** is an important result for approximate computing:
  - Programs are not tied to a specific approximate hardware model
  - Approximate hardware models are changing rapidly and specific to physical hardware

Questions?

Backup Slide

Key goals

1. Computation runs to **completion**
2. Computation yields an **acceptable** result
3. **Savings** from using approximate hardware

Optimization 1: Stable Data

- **Stable Data**: Data that is unchanged for all tasks in taskset
  - *e.g. images, data structures*
- **Optimization**: Selectively send stable data
  - with first task in taskset
  - with task following approximate machine crash
  - with task following n rejected errors
- **Reduce overhead if task contains large unchanging inputs**

```
// computes the weights for each valid pose.
taskset   calcweights(i=0; i<particles.size(); i+=1){
  compute in (
    float tpart[P_SIZE] =  (float*) particles[i],
    const float tmodel[M_SIZE] = (float*) mdl_prim,
    const char timg[I_SIZE] = (char *) img_prim,
    const int nCams = mModel->NCameras(),
    const int nBits = mModel->getBytesPerPixel(),
    const int width = mModel->getWidth(),
    const int height =mModel->getHeight()
  ) out (float tweight) {
      tweight = CalcWeight(tpart,
          tmodel, timg, nCams, width, height, nBits);
  }
```

`const` stable data annotation for inputs

Optimization 2: Abstract Output Vector (AOV)

- **Abstract Output Vector (AOV)**: Programmer defined result tuple abstraction.
- **<u>Optimization</u>**: Outlier detector performs detection on AOV.
  - AOV smaller than result tuple
  - lower dimensionality outlier detection
- **<u>Aside</u>**: Handle input dependence using AOV
- **Reduces outlier detector overhead if AOV smaller than result tuple**

```
taskset  name(int i = l; i < u; i++) {
  compute in  (d1 x1 = e1, ..., dn xn = en)
              out (o1 y1, ..., oj yj) {
    <task body>
  }
  transform out (v1, ..., vk) {
    <output abstraction>
  }
  combine { <combine body> }
}
```

`transform` block with AOV outputs `v1..vk`. `⟨output abstraction⟩` defines the transformation

AOV: Two Examples

```
transform out(float ea, float ev,float ephi)
{
  ephi=ev=ea=0; int k=0;
  for(int b=0; b < BATCH; b++){
    for(int d=0; d < NDIMS; d++, k++){
      ev += square(vel[k]);
      ea += square(acc[k]);
    }
    ephi += phi[b];
  }
}
```

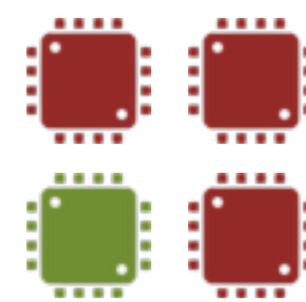
`transform` block for `water` simulation reduces output dimensionality

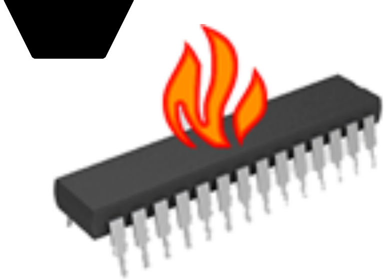
```
transform out (bweight, bp1, bp2, bp3) {
      bweight = tweight;
      bp1=tpart[3]; bp2=tpart[4]; bp3=tpart[5];
}
```

`transform` block for `bodytrack` accounts for input dependence

The world contains a lot of **approximate** hardware

- Hardware with **manufacturing defects**
- **Older**, heavily used machines
- Hardware in **aggressive** conditions
- Hardware with **protections** removed
- Novel hardware created from **immature** fabrication processes
- Hardware intentionally engineered to occasionally **produce errors** for **energy** and **performance** savings



Key Question

Can we use **approximate** hardware?

Key Question

Can we use **approximate** hardware?

Yes!

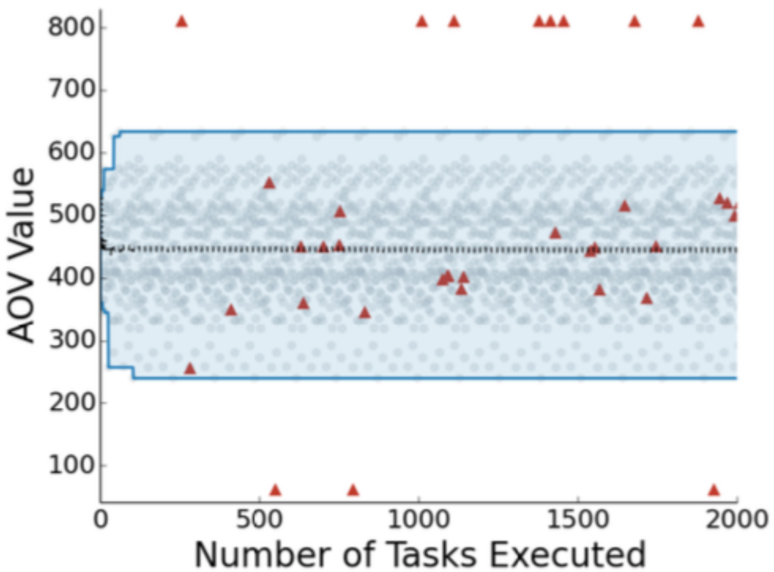
How Can we Use Approximate Hardware?

- **Just Use It:** crashes, unacceptable results
- **Static Analysis:** statically derive probabilistic error bounds
  - requires hardware specification / fault model
  - fine grain control over approximate hardware faults
  - no runtime overhead and probabilistic guarantee
- **Dynamic Systems**: adjust to faults that occur during runtime
  - weaker guarantees and runtime overhead
  - adaptive, robust

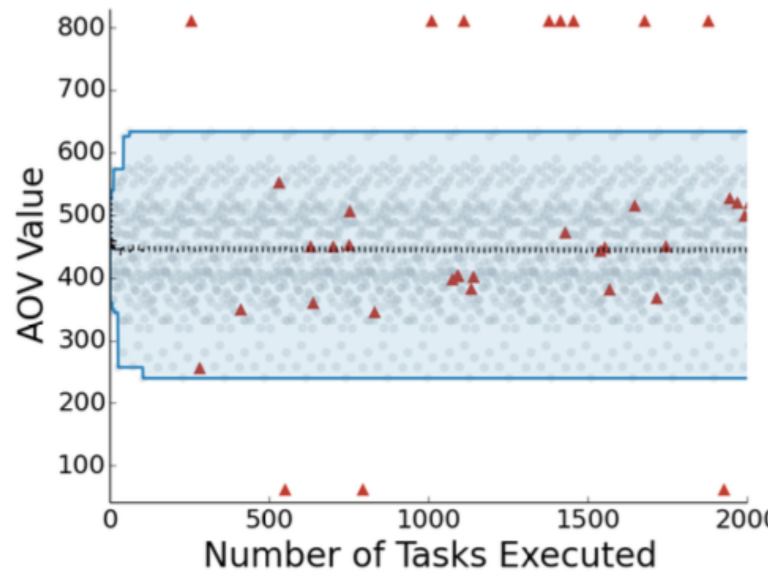
When is it acceptable for **approximations** to occur?

- **forgiving** and **critical** code and data
- if an error occurs in code or data:
  - **critical**: program failure
  - **forgiving**: different answer
- target programs that spend most of computation in **forgiving** regions

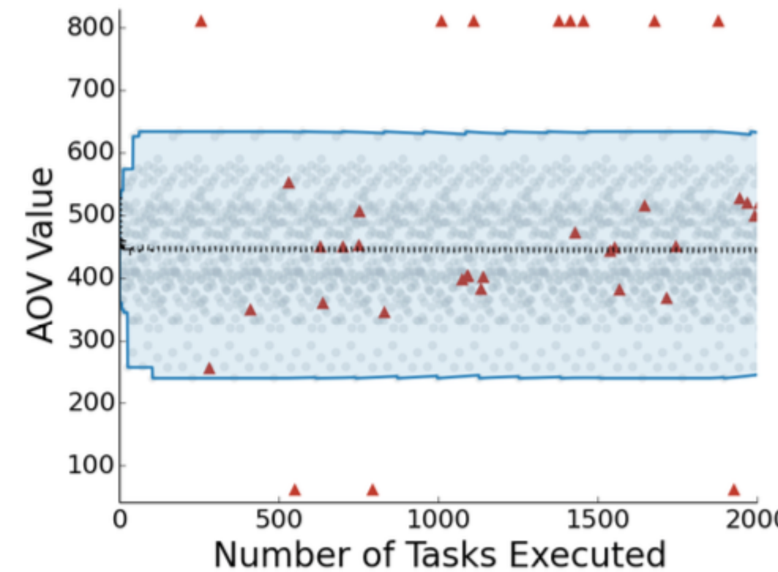
Qualitative adaptive outlier detector efficacy



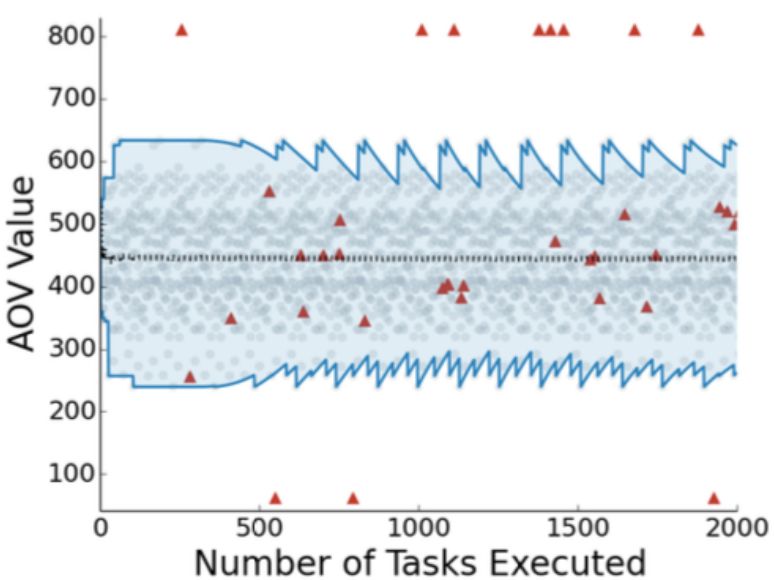
(a) 0% Target Reexecution Rate



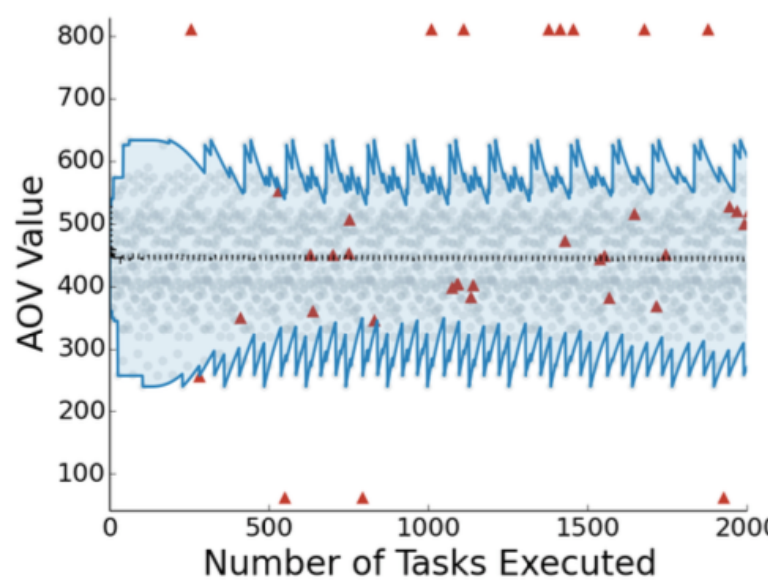
(b) 1% Target Reexecution Rate



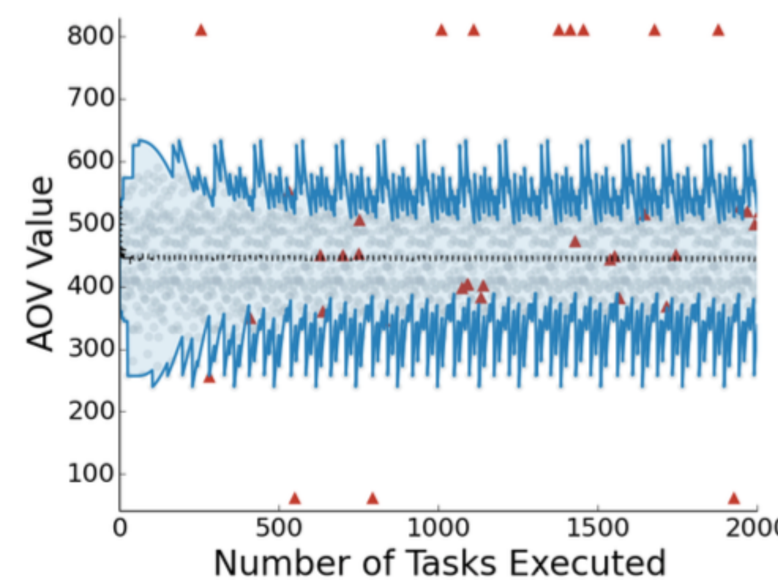
(c) 2% Target Reexecution Rate



(d) 4% Target Reexecution Rate



(e) 8% Target Reexecution Rate



(f) 16% Target Reexecution Rate