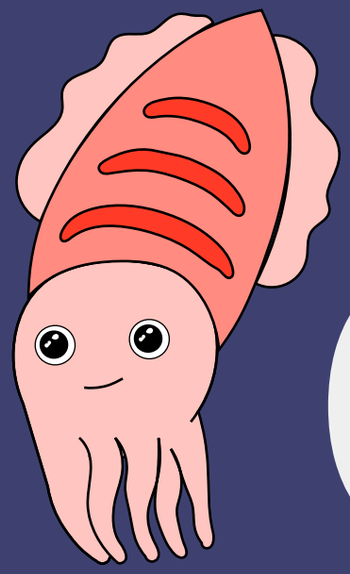


Cuttlefish

A Lightweight Primitive for Online Tuning

by Tomer Kaftan (UW), Magdalena Balazinska (UW), Alvin Cheung (UW), Johannes Gehrke (Microsoft)

Logical Operators have multiple physical Operators…
The system should automatically choose!

Pages in category "Join algorithms"

The following 5 pages are in this category, out of 5 total.

B

- Block nested loop

H

- Hash join

N

- Nested loop join

S

- Sort-merge join
- Symmetric Hash Join

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$ (simple partition) or $n + k$ (three-way partition and equal keys)	$n \log n$	n^2	Depends on the algorithm and implementation. $\log n$ worst case.	Typical in-place sort is not; stable versions exist.	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space.[citation needed][clarification needed]
Merge sort	$n \log n$	$n \log n$	$n \log n$	n Hybrid block merge sort is $O(1)$ mem.	Yes	Merging	Highly parallelizable (up to $O(\log n)$ using the Three Hungarians' Algorithm[citation needed]) or, more practically, Cole's parallel merge sort for processing large amounts of data.
In-place merge sort	—	—	$n \log^2 n$ See above for hybrid, merge $n \log n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.[citation needed]
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	$O(nk)$, in the worst case over sequences that have k inversions.
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with $O(n)$ extra space, for example using lists.[citation needed]
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	Makes n comparisons when the data is already sorted or reverse sorted.
Cubesort	n	$n \log n$	$n \log n$	n	Yes	Insertion	Makes n comparisons when the data is already sorted or reverse sorted.
Shellsort	$n \log n$	$n \log^2 n$ or $n^{4/3}$	Depends on gap sequence; best known is $n \log^2 n$	1	No	Insertion	Small code size, no use of call stack, reasonably fast, useful where memory is at a premium such as embedded and older mainframe applications. Best case $n \log n$ and worst case $n \log n$ can be achieved together. With best case $n \log n$, best worst case is n^2.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Binary tree sort	$n \log n$	$n \log n$	$n \log n$ (balanced)	n	Yes	Insertion	When using a self-balancing binary search tree.
Cycle sort	n^2	n^2	n^2	1	No	Insertion	In-place with theoretically optimal number of writes.
Library sort	n	$n \log n$	n^2	n	Yes	Insertion	
Patience sorting	n	—	$n \log n$	n	No	Insertion & Selection	Finds all the longest increasing subsequences in $O(n \log n)$.
Strand sort	n	n^2	n^2	n	Yes	Selection	
Cocktail sort	n	n^2	n^2	1	Yes	Exchanging	

Pages in category "External memory algorithms"

The following 3 pages are in this category, out of 3 total. Thi

C

- Cache-oblivious distribution sort

E

- External sorting

F

- Funnelsort

Logical Operators have multiple physical Operators…
The system should automatically choose!

(Some) Prior Work on Query Optimization

(Some) Prior Work on Query Optimization

- Static Query Optimizers
  - cardinality & selectivity estimation , heuristics, cost models

(Some) Prior Work on Query Optimization

- Static Query Optimizers
  - cardinality & selectivity estimation , heuristics, cost models
- Adaptive Query Optimization
  - Query re-optimization (update cardinality & selectivity estimates)
  - Adaptive operators (scans, aggregates, etc.)
  - Eddies & adaptive tuple routing (operator reordering)

These work great, BUT...

These work great, BUT…

- Designing good query optimizers takes time!

These work great, BUT…

- Designing good query optimizers takes time!
- Requires deep knowledge of the operators and significant development effort

These work great, BUT…

- Designing good query optimizers takes time!
- Requires deep knowledge of the operators and significant development effort
- Spark SQL took 2 years to go from heuristics-based optimization to cost-based optimization! [1]

[1] http://databricks.com/blog/2017/08/31/cost-based-optimizer-in-apache-spark-2- 2.html

These work great, BUT…

- Designing good query optimizers takes time!
- Requires deep knowledge of the operators and significant development effort
- Spark SQL took 2 years to go from heuristics-based optimization to cost-based optimization! [1]
- Existing adaptive approaches just push the development overhead to physical execution

[1] http://databricks.com/blog/2017/08/31/cost-based-optimizer-in-apache-spark-2- 2.html

These work great, BUT…

- Designing good query optimizers takes time!
- Requires deep knowledge of the operators and significant development effort
- Spark SQL took 2 years to go from heuristics-based optimization to cost-based optimization! [1]
- Existing adaptive approaches just push the development overhead to physical execution
- Modern data processing applications involve diverse, sophisticated operators, not just relational operators!

[1] http://databricks.com/blog/2017/08/31/cost-based-optimizer-in-apache-spark-2- 2.html

Motivating Workload

Motivating Workload

→ “A Cuttlefish pretending to be a rock”

*Image Sourced from https://www.flickr.com/photos/silkebaron/32001215104

Motivating Workload

→ "A Cuttlefish pretending to be a rock"

*Image Sourced from https://www.flickr.com/photos/silkebaron/32001215104

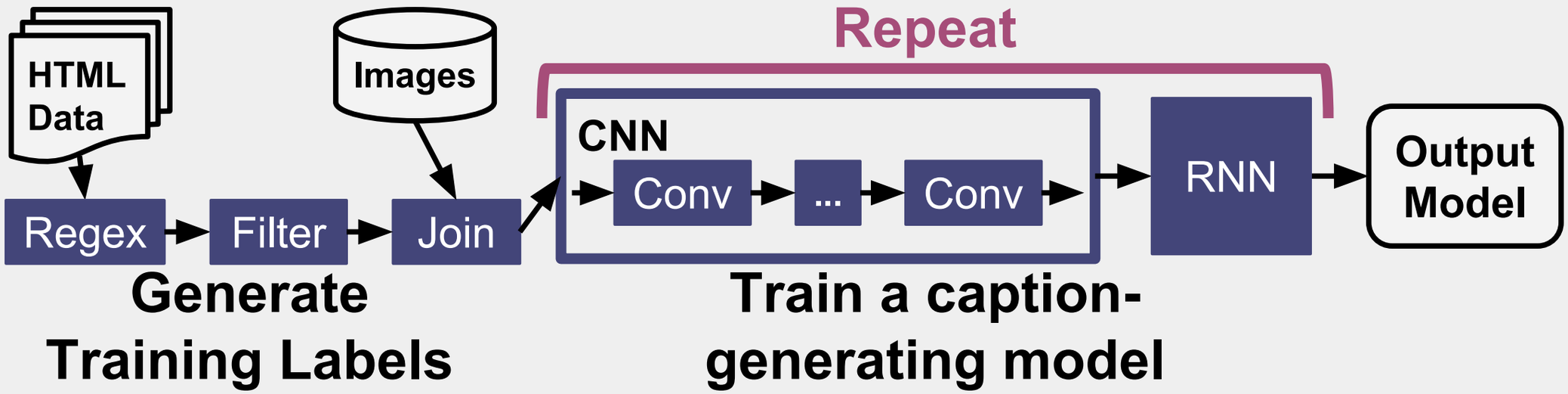
Generate Training Data from:

flickr

imgur

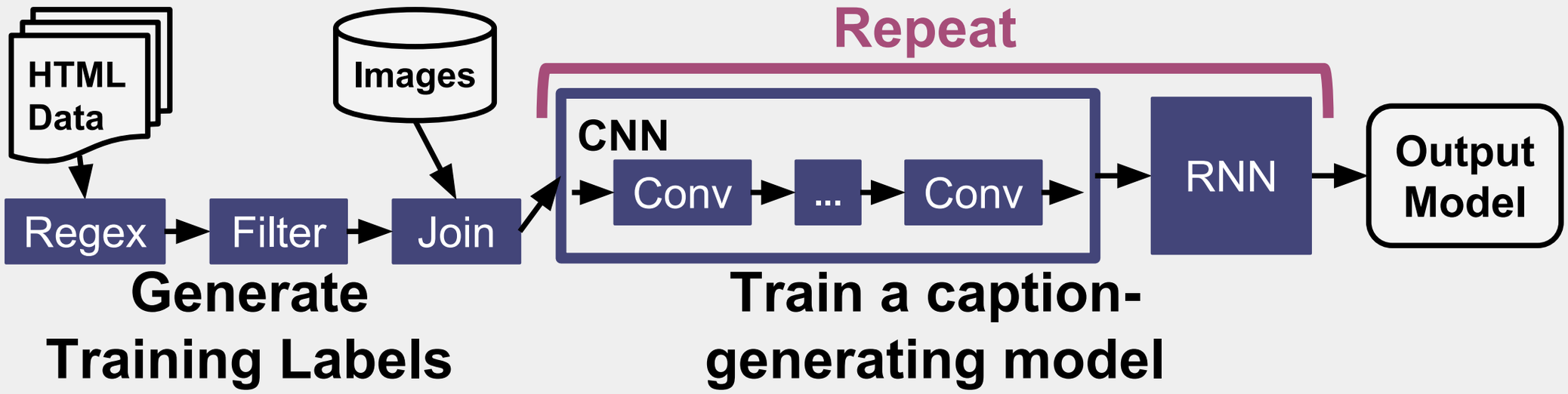
etc.

Motivating Workload



*caption-generating model portion of the logical plan inspired by: Xu et al. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. ICML 2015

Motivating Workload



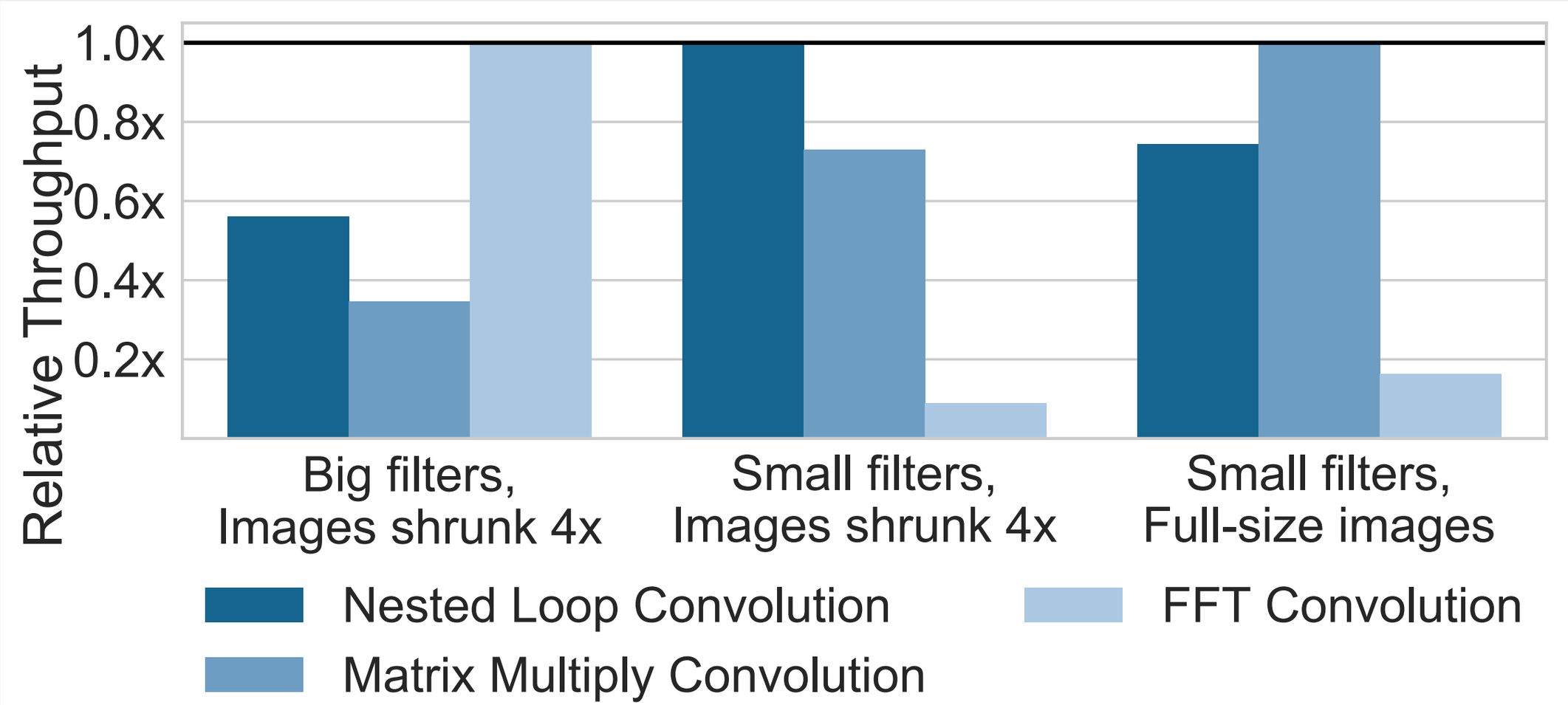
Diverse, sophisticated operators, with multiple physical alternatives!

*caption-generating model portion of the logical plan inspired by: Xu et al. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. ICML 2015

Example Operator: Convolution

Example Operator: Convolution

Tested 3 convolution algorithms on 8000 Flickr images



Can we optimize without a full-fledged optimizer?

Prior Work: Tuning Black-box Operators

Prior Work: Tuning Black-box Operators

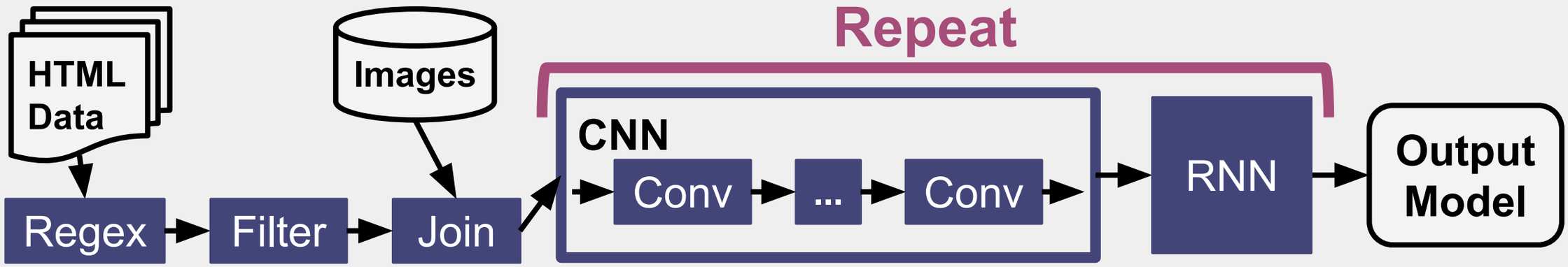
- Offline Autotuning
  - Searches through arbitrary physical plan spaces
  - Requires representative workloads
  - Offline training time

Prior Work: Tuning Black-box Operators

- Offline Autotuning
  - Searches through arbitrary physical plan spaces
  - Requires representative workloads
  - Offline training time

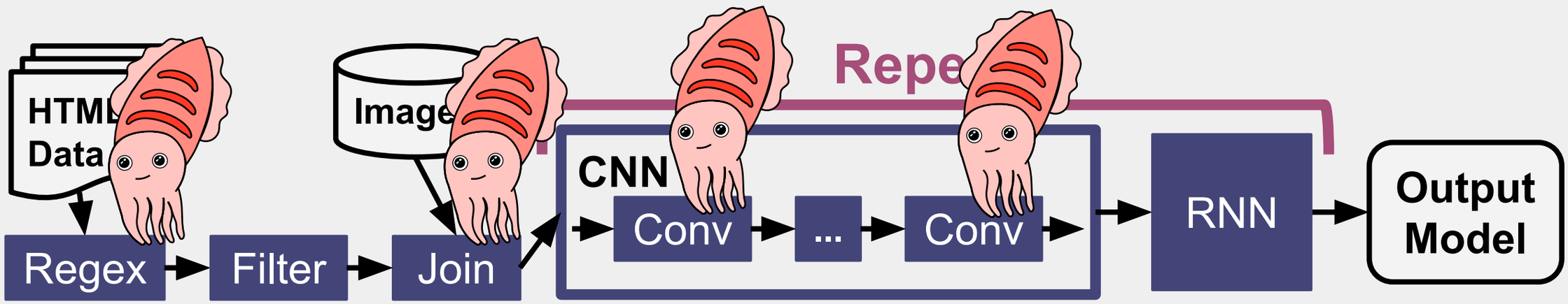
- Micro Adaptivity in Vectorwise
  - Reinforcement learning chooses physical flavors of black-box vectorized operators
  - Limited to vectorized operators, does not explore multi-core settings

Cuttlefish:
A Lightweight Primitive for Online Tuning



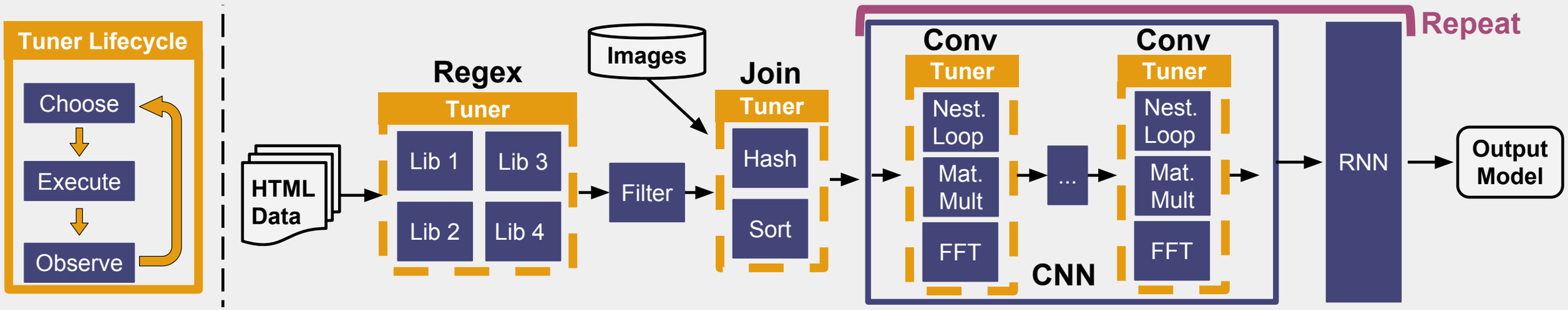
Workload developer (or the query optimizer) inserts calls to Cuttlefish's API to pick physical operators ***during execution***

Cuttlefish:
A Lightweight Primitive for Online Tuning



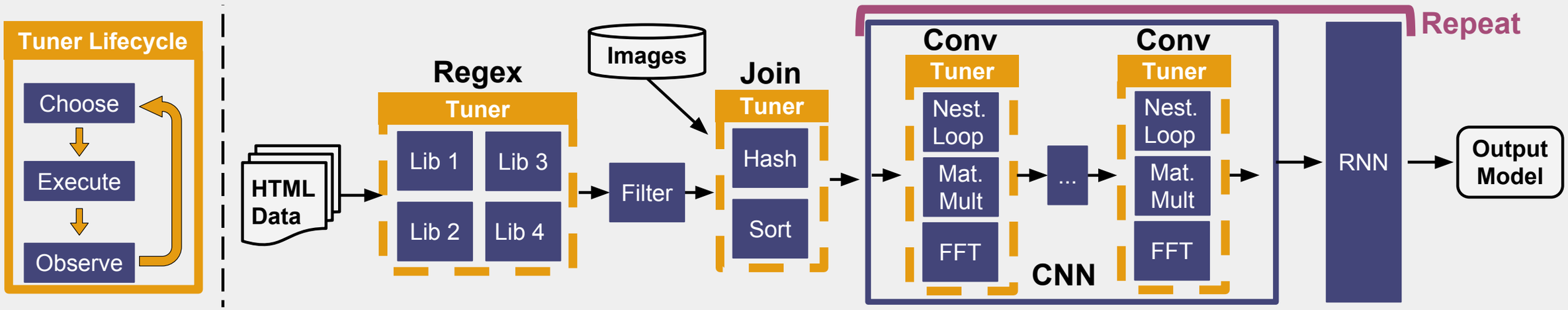
Workload developer (or the query optimizer) inserts calls to Cuttlefish's API to pick physical operators ***during execution***

Cuttlefish:
A Lightweight Primitive for Online Tuning



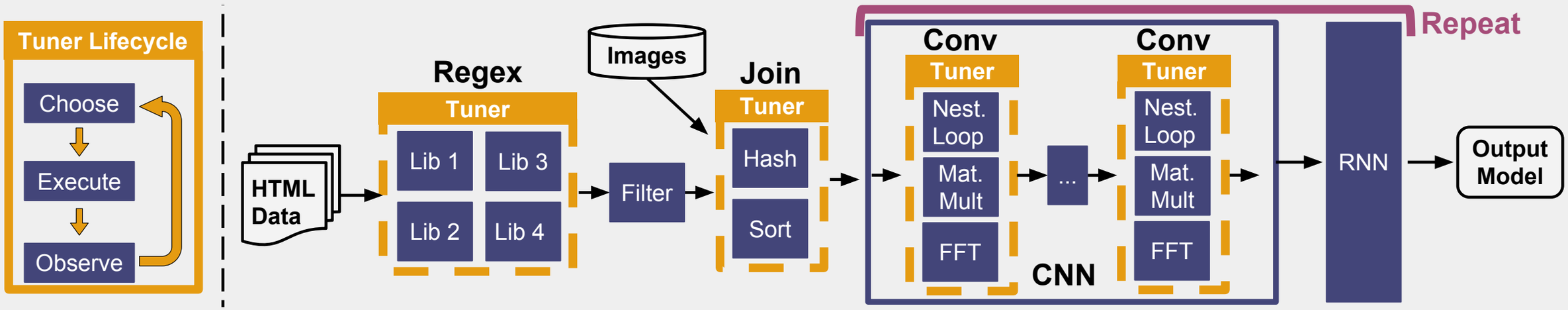
Workload developer (or the query optimizer) inserts calls to Cuttlefish's API to pick physical operators ***during execution***

Cuttlefish:
A Lightweight Primitive for Online Tuning



Developer maps tuning rounds to the execution model of each operator:

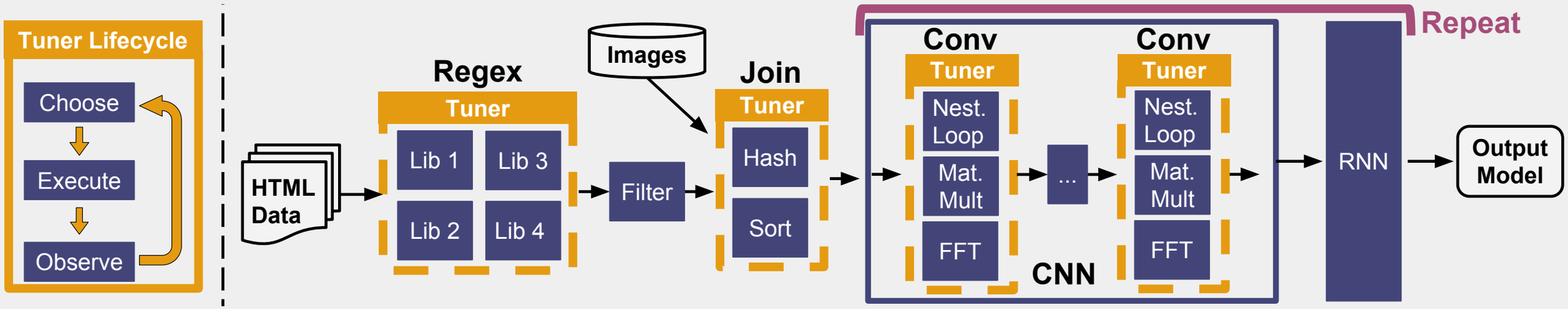
Cuttlefish:
A Lightweight Primitive for Online Tuning



Developer maps tuning rounds to the execution model of each operator:

- Regex: One round per HTML Doc

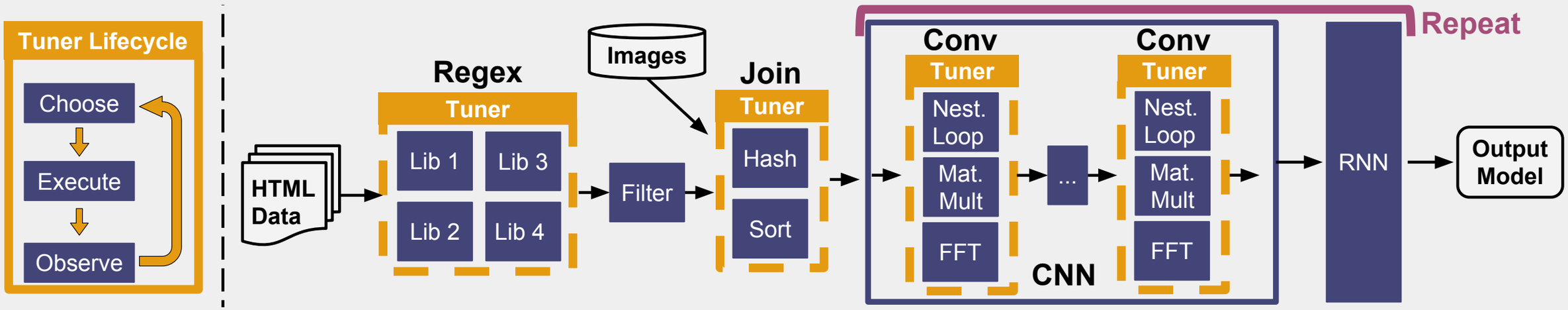
Cuttlefish:
A Lightweight Primitive for Online Tuning



Developer maps tuning rounds to the execution model of each operator:

- Regex: One round per HTML Doc
- Convolve: One round per image

Cuttlefish:
A Lightweight Primitive for Online Tuning



Developer maps tuning rounds to the execution model of each operator:

- Regex: One round per HTML Doc
- Convolve: One round per image
- Parallel Distributed Join: One round per partition

Cuttlefish

I. Problem & Motivation

II. The Cuttlefish API

III. Bandit-based Online Tuning

IV. Distributed Tuning Approach

V. Contextual Tuning

VI. Handling Nonstationary Settings

VII. Other Operators

VIII. Conclusion

The Cuttlefish Primitive

The Cuttlefish Primitive

1. Construct a tuner (from a set of choices)

The Cuttlefish Primitive

1. Construct a tuner (from a set of choices)

2. Tuner.choose (pick one of the choices)

The Cuttlefish Primitive

1. Construct a tuner (from a set of choices)

2. Tuner.choose (pick one of the choices)

3. Tuner.observe (observe a reward for a choice)

The Cuttlefish Primitive

1. Construct a tuner (from a set of choices)

2. Tuner.choose (pick one of the choices)

3. Tuner.observe (observe a reward for a choice)

Cuttlefish tuners maximize the total reward after multiple choose-observe tuning rounds

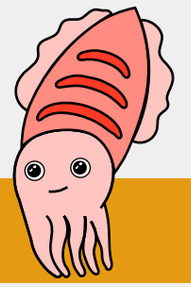
Tuning Convolution with Cuttlefish

Tuning Convolution with Cuttlefish

```
def loopConvolve(image, filters): …
def fftConvolve(image, filters): …
def mmConvolve(image, filters): …
```

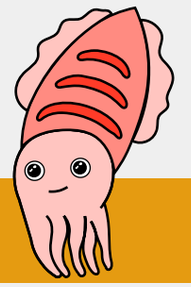
Tuning Convolution with Cuttlefish

```
def loopConvolve(image, filters): …
def fftConvolve(image, filters): …
def mmConvolve(image, filters): …
tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])
```



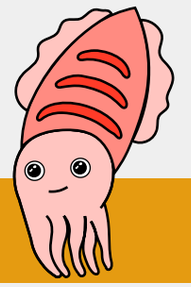
Tuning Convolution with Cuttlefish

```
def loopConvolve(image, filters): …
def fftConvolve(image, filters): …
def mmConvolve(image, filters): …
tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])
for image, filters in convolutions:
```



Tuning Convolution with Cuttlefish

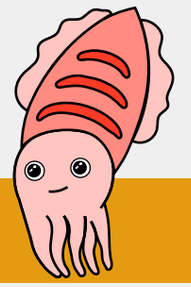
```
def loopConvolve(image, filters): …
def fftConvolve(image, filters): …
def mmConvolve(image, filters): …
tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])
```



```
for image, filters in convolutions:
  convolve, token = tuner.choose()
```

Tuning Convolution with Cuttlefish

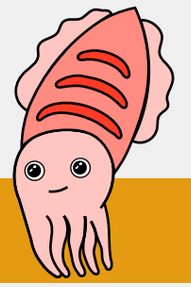
```
def loopConvolve(image, filters): …
def fftConvolve(image, filters): …
def mmConvolve(image, filters): …
tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])
```



```
for image, filters in convolutions:
  convolve, token = tuner.choose()
  start = now()
  result = convolve(image, filters)
  elapsedTime = now() - start
```

Tuning Convolution with Cuttlefish

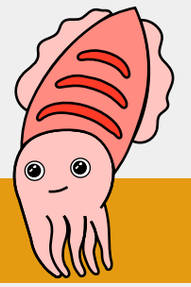
```
def loopConvolve(image, filters): …
def fftConvolve(image, filters): …
def mmConvolve(image, filters): …
tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])
```



```
for image, filters in convolutions:
  convolve, token = tuner.choose()
  start = now()
  result = convolve(image, filters)
  elapsedTime = now() - start
  reward = computeReward(elapsedTime)
```

Tuning Convolution with Cuttlefish

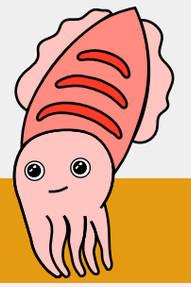
```
def loopConvolve(image, filters): …
def fftConvolve(image, filters): …
def mmConvolve(image, filters): …
tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])
```



```
for image, filters in convolutions:
  convolve, token = tuner.choose()
  start = now()
  result = convolve(image, filters)
  elapsedTime = now() - start
  reward = computeReward(elapsedTime)
  tuner.observe(token, reward)
```

Tuning Convolution with Cuttlefish

```
def loopConvolve(image, filters): …
def fftConvolve(image, filters): …
def mmConvolve(image, filters): …
tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])
```



```
for image, filters in convolutions:
  convolve, token = tuner.choose()
  start = now()
  result = convolve(image, filters)
  elapsedTime = now() - start
  reward = computeReward(elapsedTime)
  tuner.observe(token, reward)
  output result
```

Cuttlefish

I. Problem & Motivation

II. The Cuttlefish API

III. Bandit-based Online Tuning

IV. Distributed Tuning Approach

V. Contextual Tuning

VI. Handling Nonstationary Settings

VII. Other Operators

VIII. Conclusion

Approach: Tuning

Approach: Tuning

Multi-armed Bandit Problem

Approach: Tuning

Multi-armed Bandit Problem

- K possible choices (called arms)

Approach: Tuning

Multi-armed Bandit Problem

- K possible choices (called arms)
- Arms have unknown reward distributions

Approach: Tuning

Multi-armed Bandit Problem

- K possible choices (called arms)
- Arms have unknown reward distributions
- At each round: select an Arm and observe a reward

Approach: Tuning

Multi-armed Bandit Problem

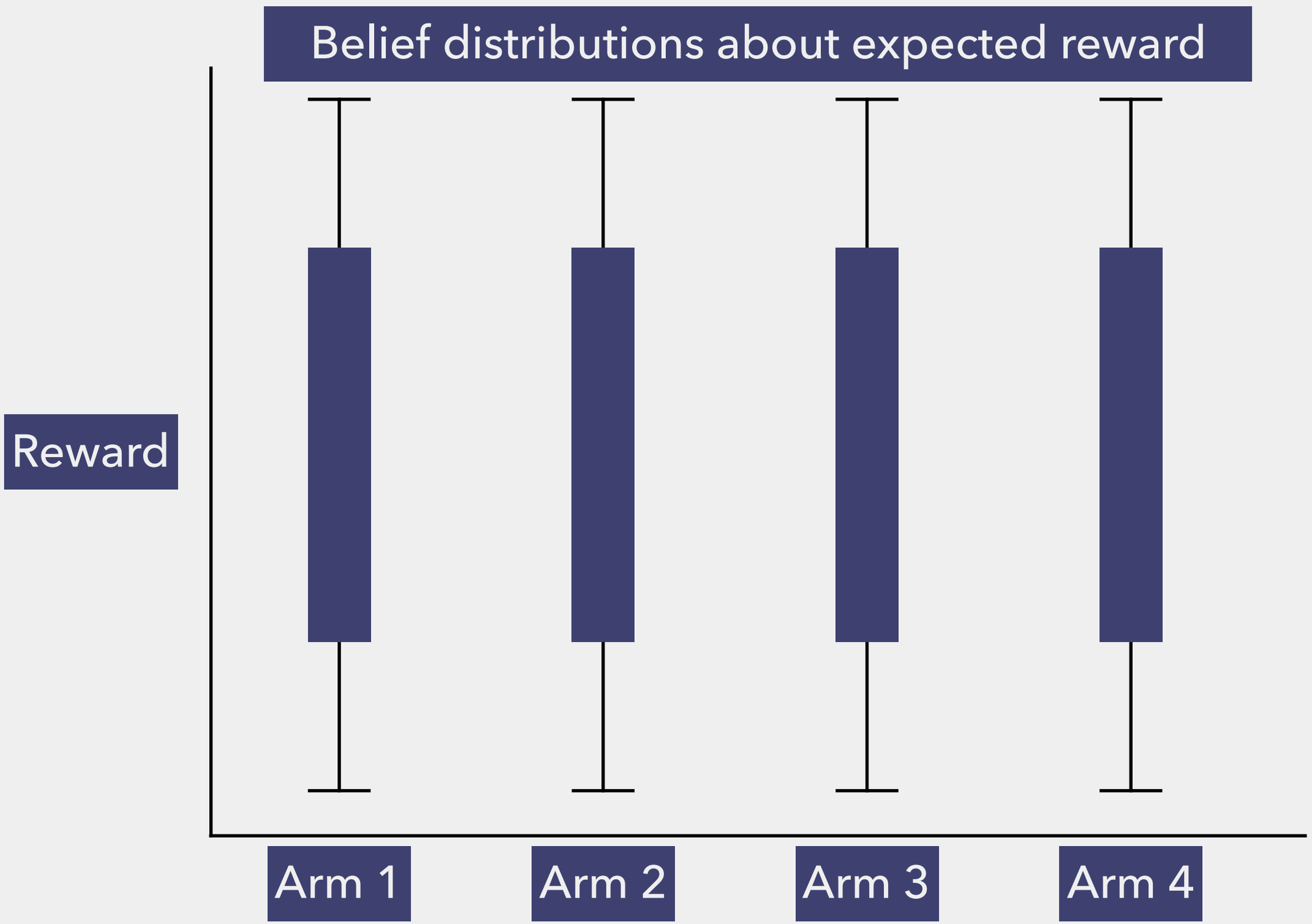
- K possible choices (called arms)
- Arms have unknown reward distributions
- At each round: select an Arm and observe a reward

Goal: Maximize Cumulative Reward

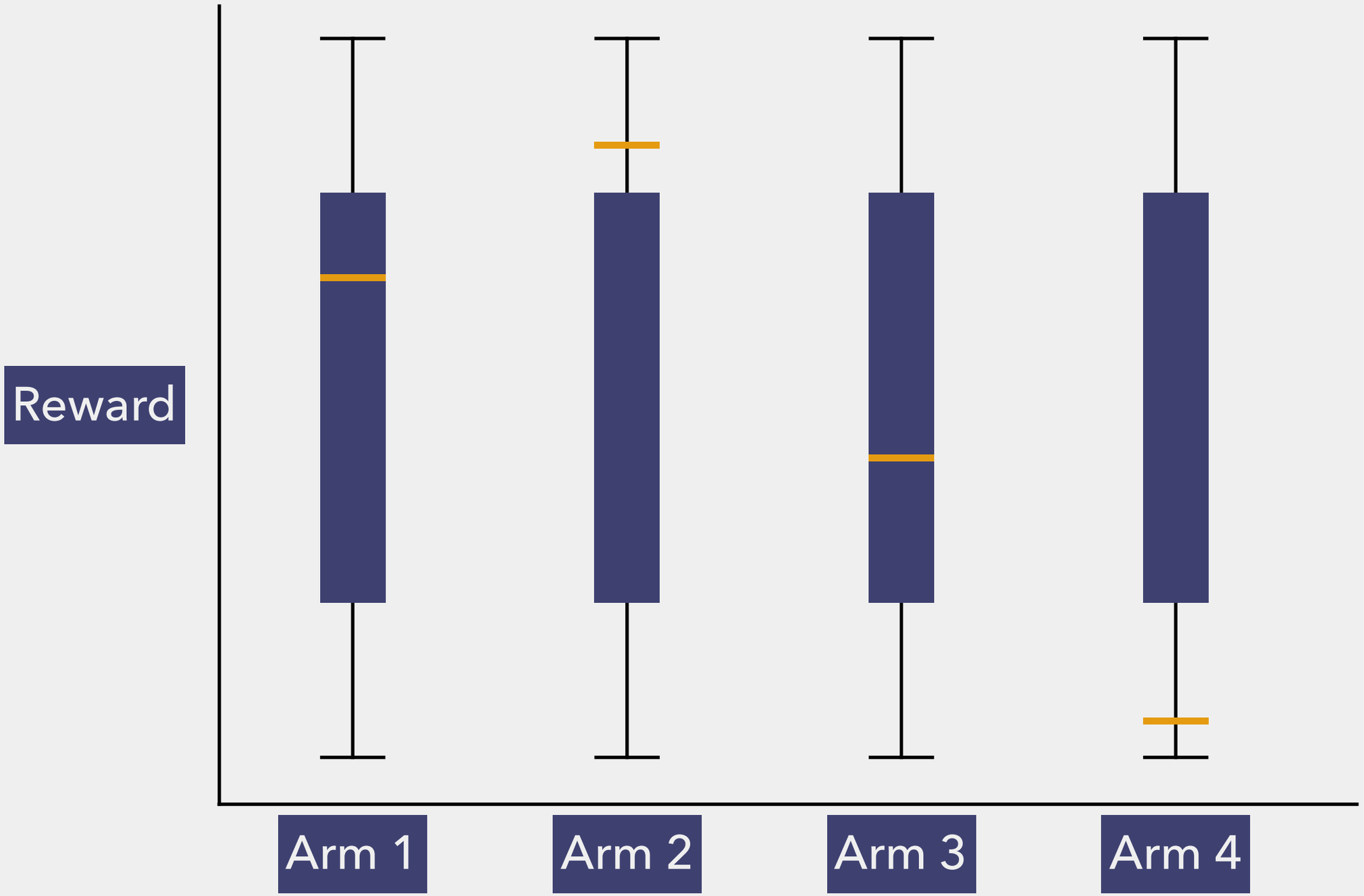
(by balancing exploration & exploitation)

Thompson Sampling

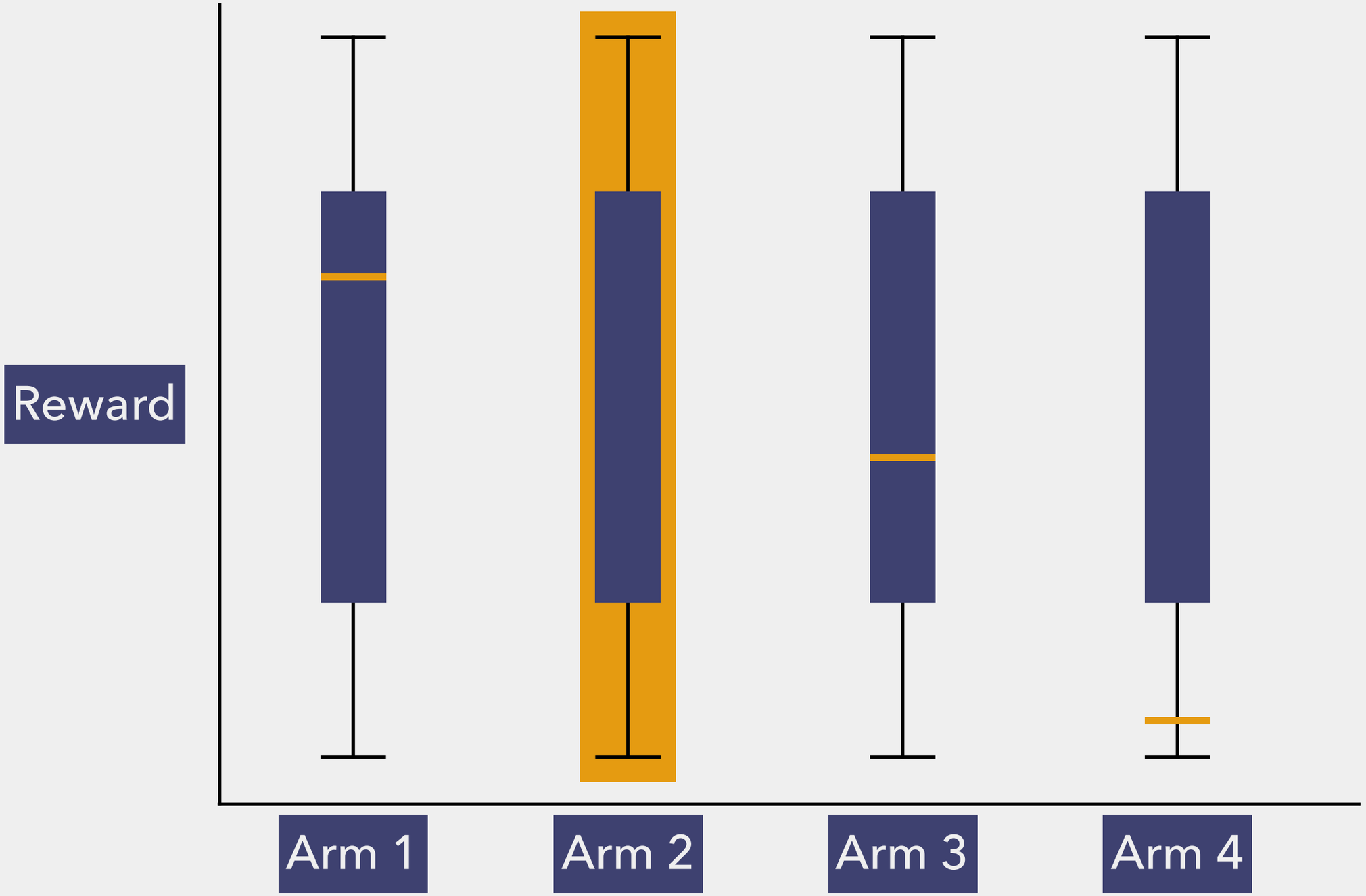
Thompson Sampling



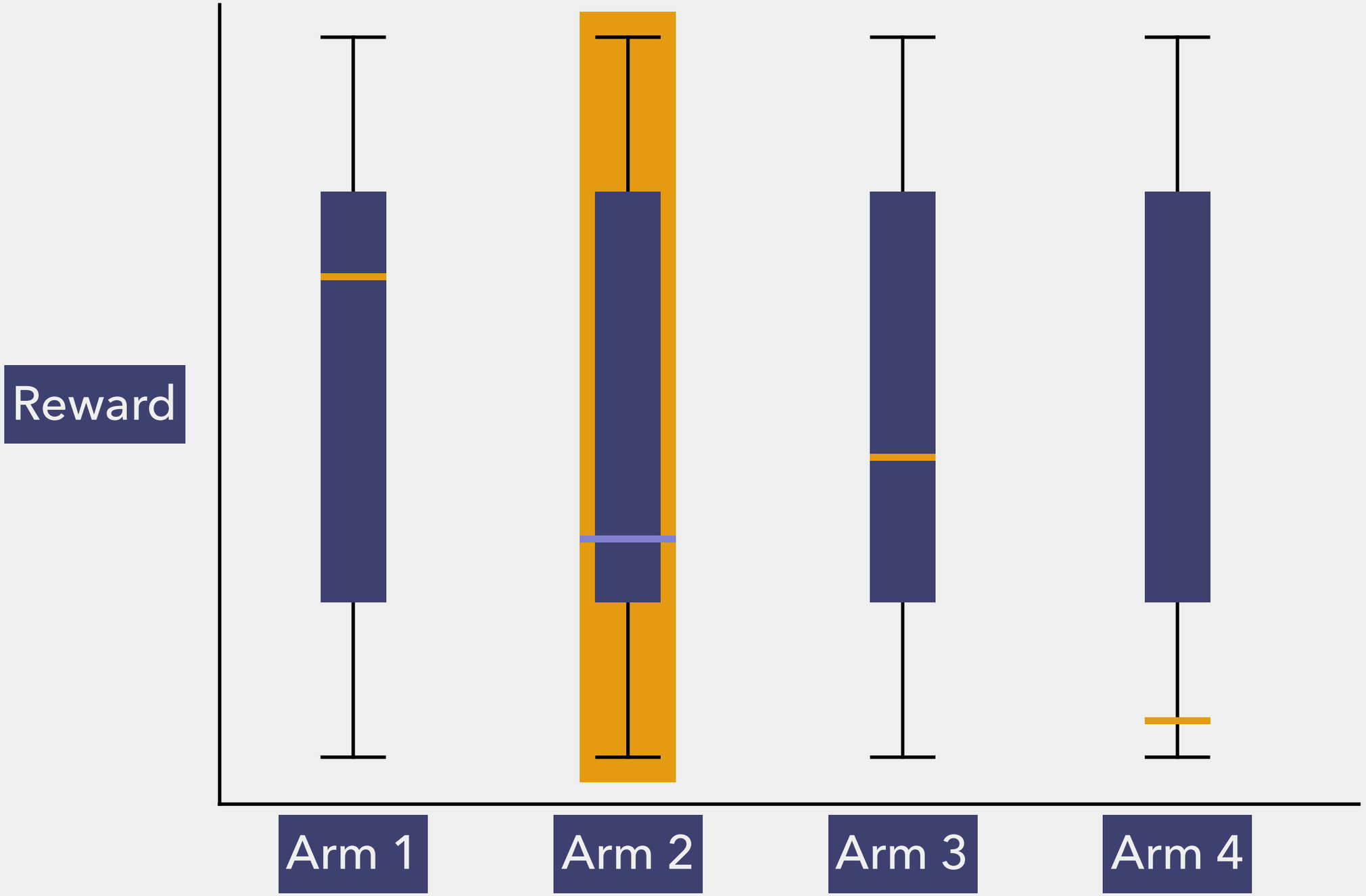
Thompson Sampling



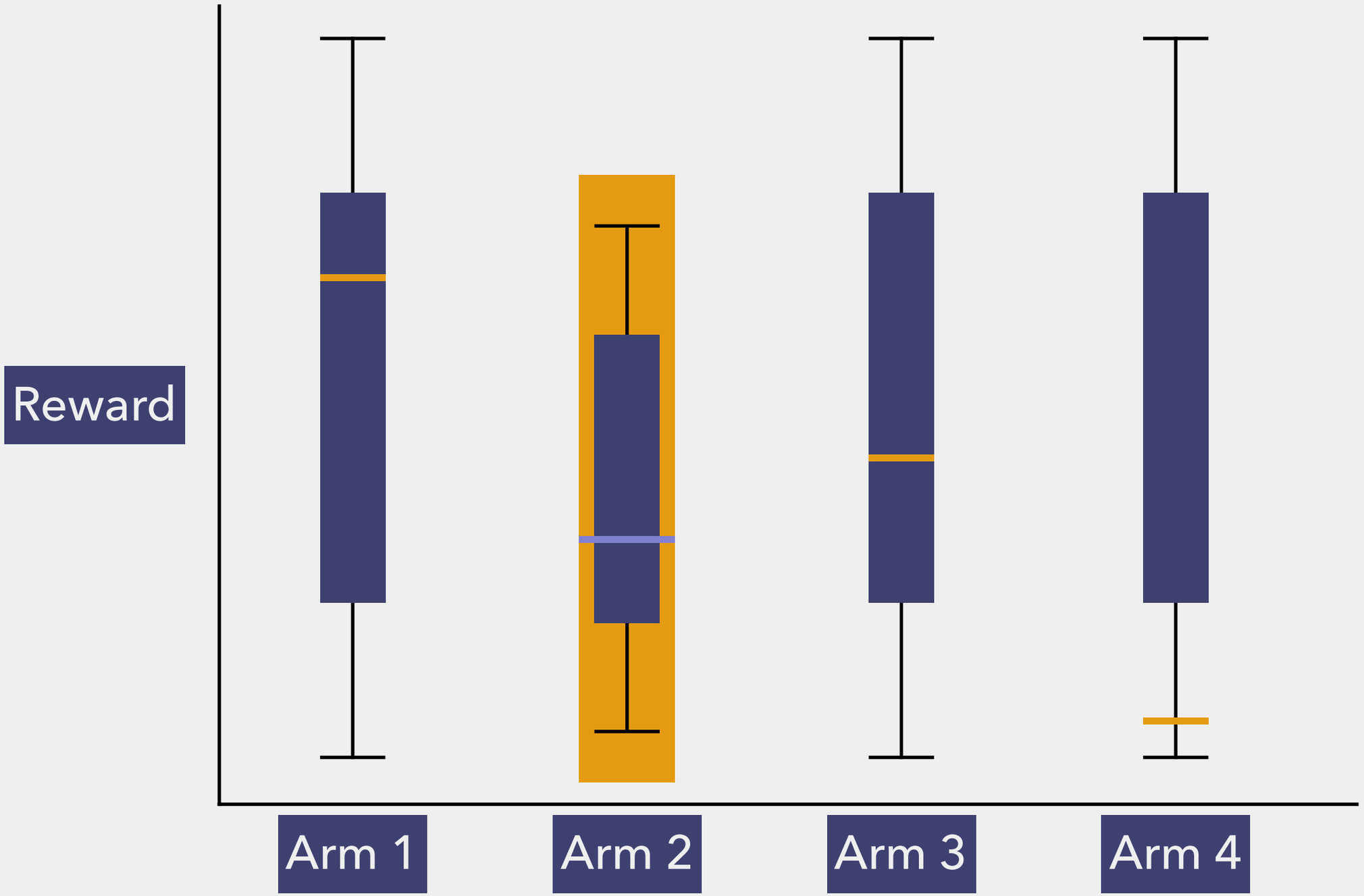
Thompson Sampling



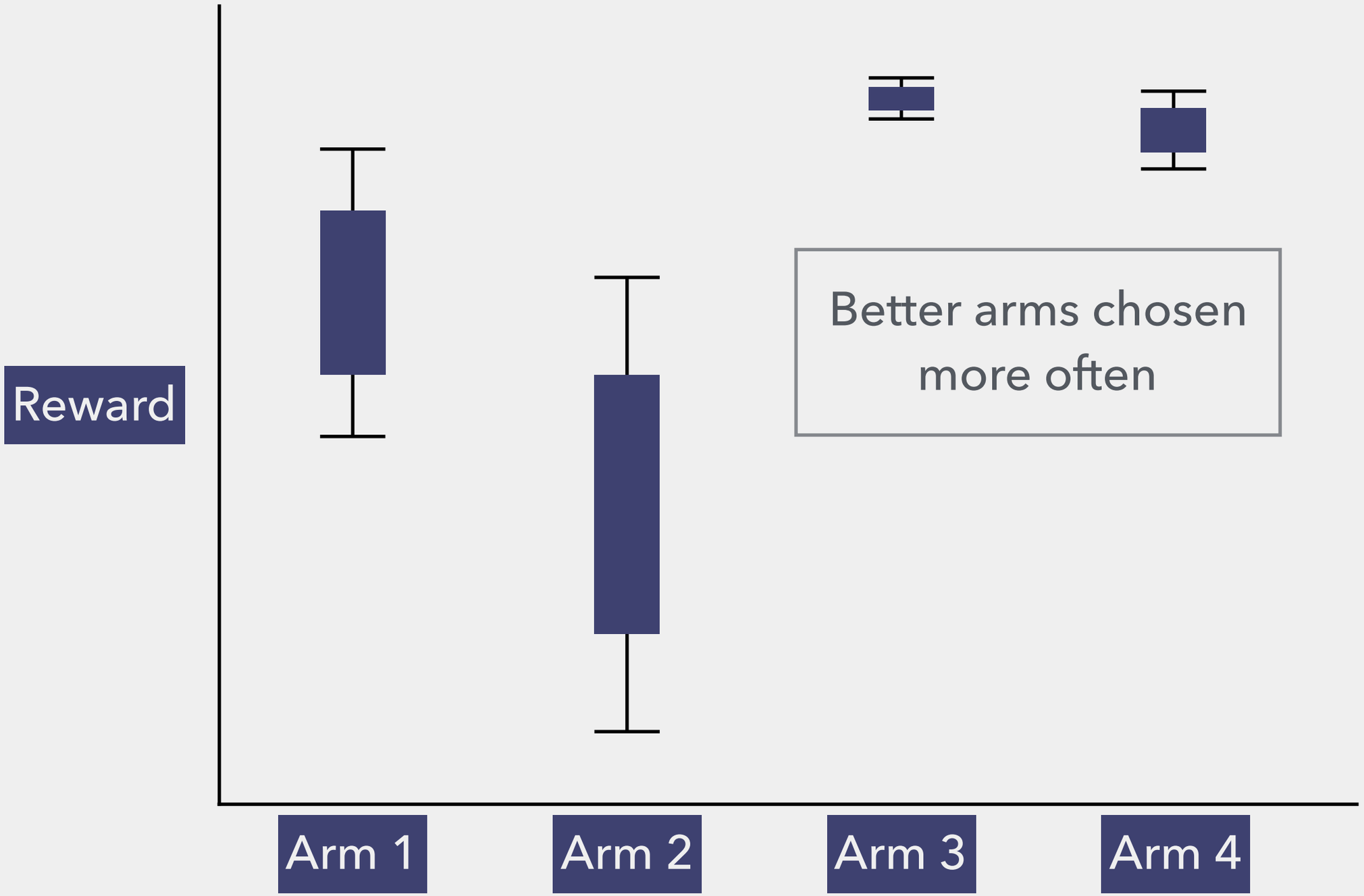
Thompson Sampling



Thompson Sampling



Thompson Sampling



Thompson Sampling

Thompson Sampling

- Gaussian runtimes with initially unknown means and variances

Thompson Sampling

- Gaussian runtimes with initially unknown means and variances
- Belief distributions form t-distributions
- Depend only on sample mean, variance, count

Thompson Sampling

- Gaussian runtimes with initially unknown means and variances
- Belief distributions form t-distributions
  - Depend only on sample mean, variance, count
- No meta-parameters, yet works well for diverse operators

Thompson Sampling

- Gaussian runtimes with initially unknown means and variances
- Belief distributions form t-distributions
  - Depend only on sample mean, variance, count
- No meta-parameters, yet works well for diverse operators
- Constant memory overhead, 0.03 ms per tuning round

Convolution Evaluation

Convolution Evaluation

- Prototype in Apache Spark

Convolution Evaluation

- Prototype in Apache Spark
- Tune between three convolution algorithms (Nested Loops, FFT, or Matrix Multiply)
    - Reward: -1*elapsedTime (maximizes throughput)

Convolution Evaluation

- Prototype in Apache Spark
- Tune between three convolution algorithms (Nested Loops, FFT, or Matrix Multiply)
    - Reward: -1*elapsedTime (maximizes throughput)
- Convolve 8000 Flickr images with sets of filters (~32gb)
    - Vary number & size of filters

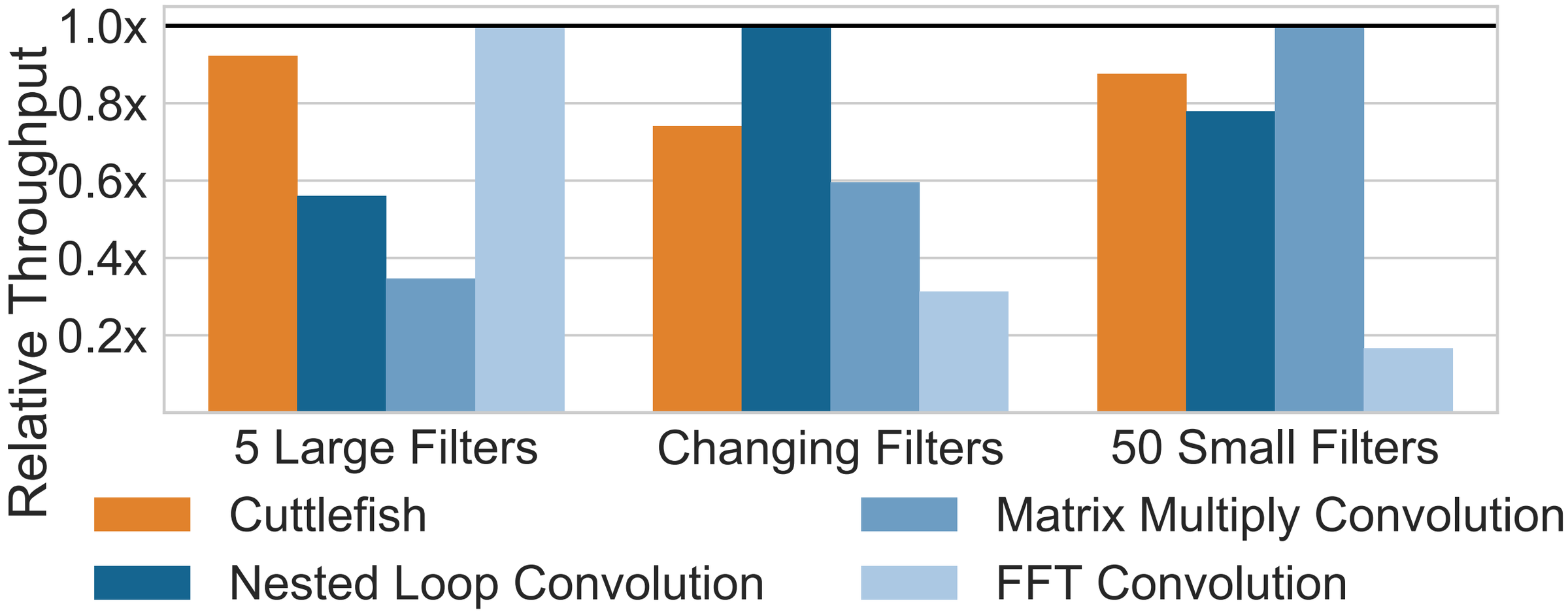
Convolution Evaluation

- Prototype in Apache Spark
- Tune between three convolution algorithms (Nested Loops, FFT, or Matrix Multiply)
    - Reward: -1*elapsedTime (maximizes throughput)
- Convolve 8000 Flickr images with sets of filters (~32gb)
    - Vary number & size of filters
- Run on an 8-node (AWS EC2 4-core r3.xlarge) cluster.
    - 32 total cores, ~252 images per core

Convolution Evaluation

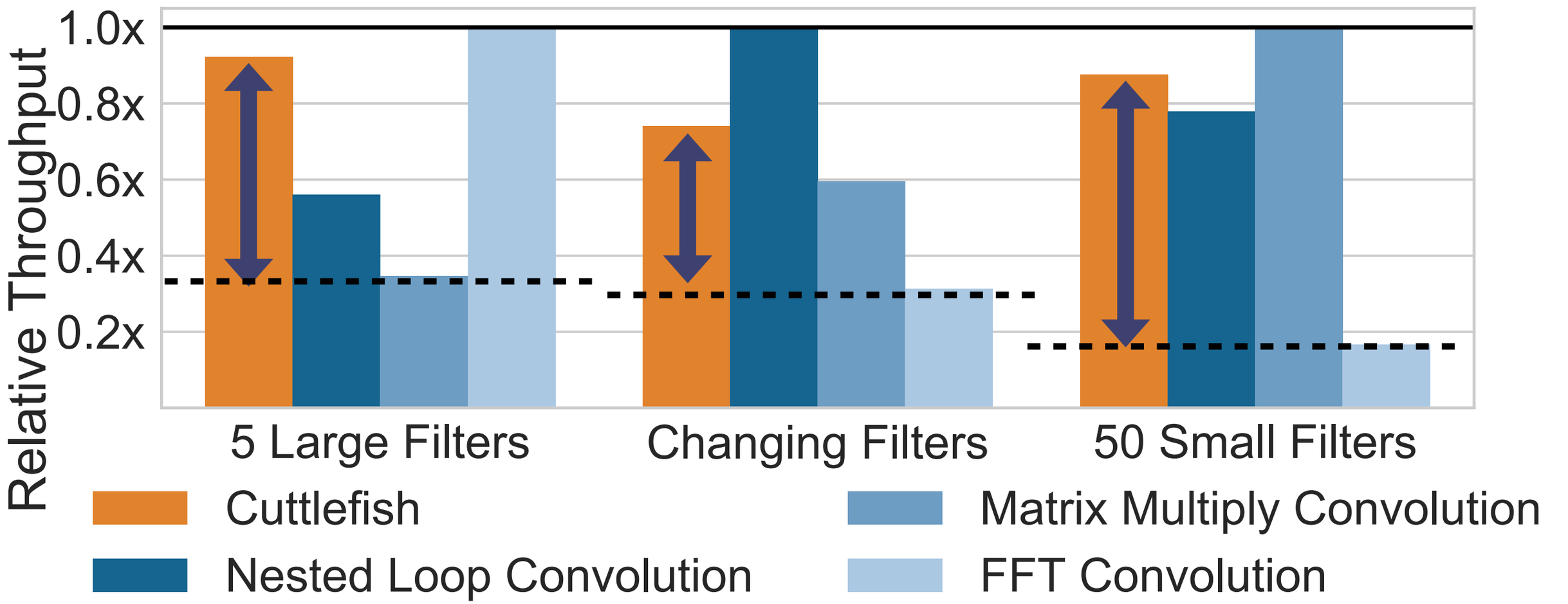
- Prototype in Apache Spark
- Tune between three convolution algorithms (Nested Loops, FFT, or Matrix Multiply)
    - Reward: -1*elapsedTime (maximizes throughput)
- Convolve 8000 Flickr images with sets of filters (~32gb)
    - Vary number & size of filters
- Run on an 8-node (AWS EC2 4-core r3.xlarge) cluster.
    - 32 total cores, ~252 images per core
- *Very* compute intensive
    - (Some configs up to 45 min on a single node)

Convolution Results



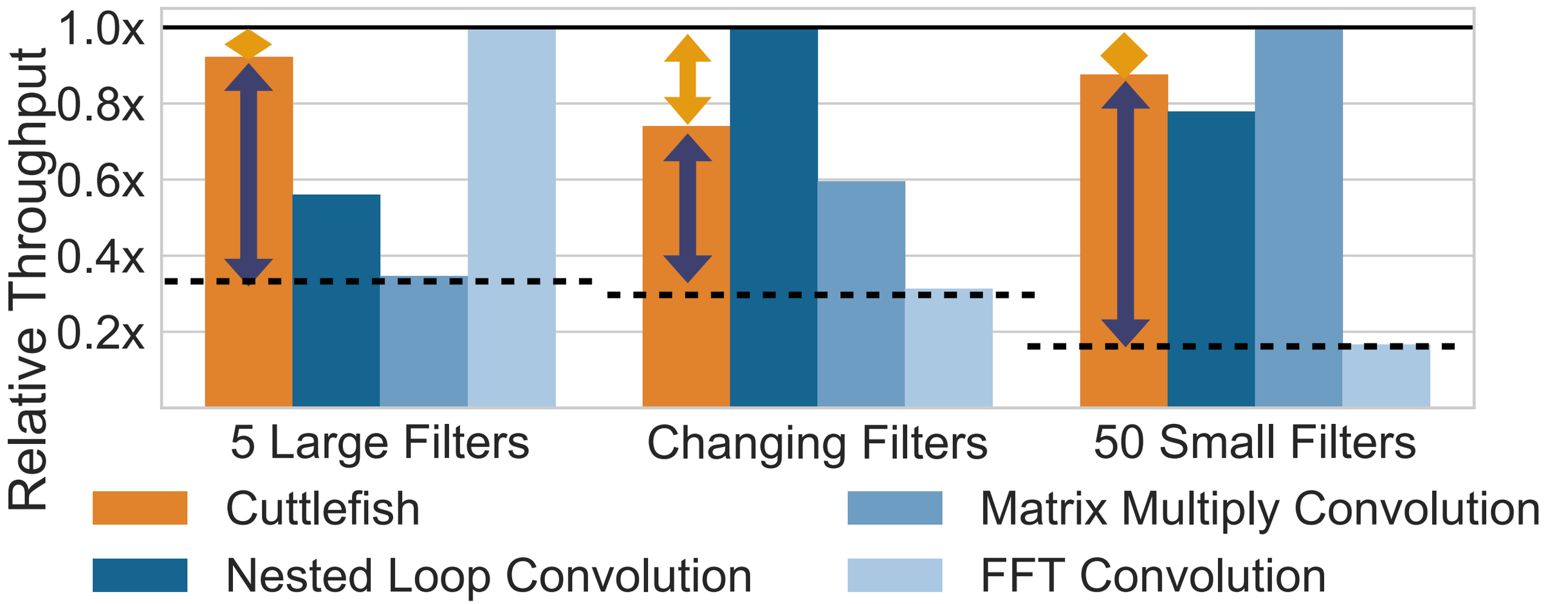
Relative throughput normalized against the highest-throughput algorithm

Convolution Results



Relative throughput normalized against the highest-throughput algorithm

Convolution Results



Relative throughput normalized against the highest-throughput algorithm

Cuttlefish

I. Problem & Motivation

II. The Cuttlefish API

III. Bandit-based Online Tuning

IV. Distributed Tuning Approach

V. Contextual Tuning

VI. Handling Nonstationary Settings

VII. Other Operators

VIII. Conclusion

Challenges in Distributed Tuning

Challenges in Distributed Tuning

1. Choosing and observing occur throughout a cluster
    - To maximize learning, need to communicate

Challenges in Distributed Tuning

1. Choosing and observing occur throughout a cluster
    - To maximize learning, need to communicate
2. Synchronization & communication overheads

Challenges in Distributed Tuning

1. Choosing and observing occur throughout a cluster
    - To maximize learning, need to communicate

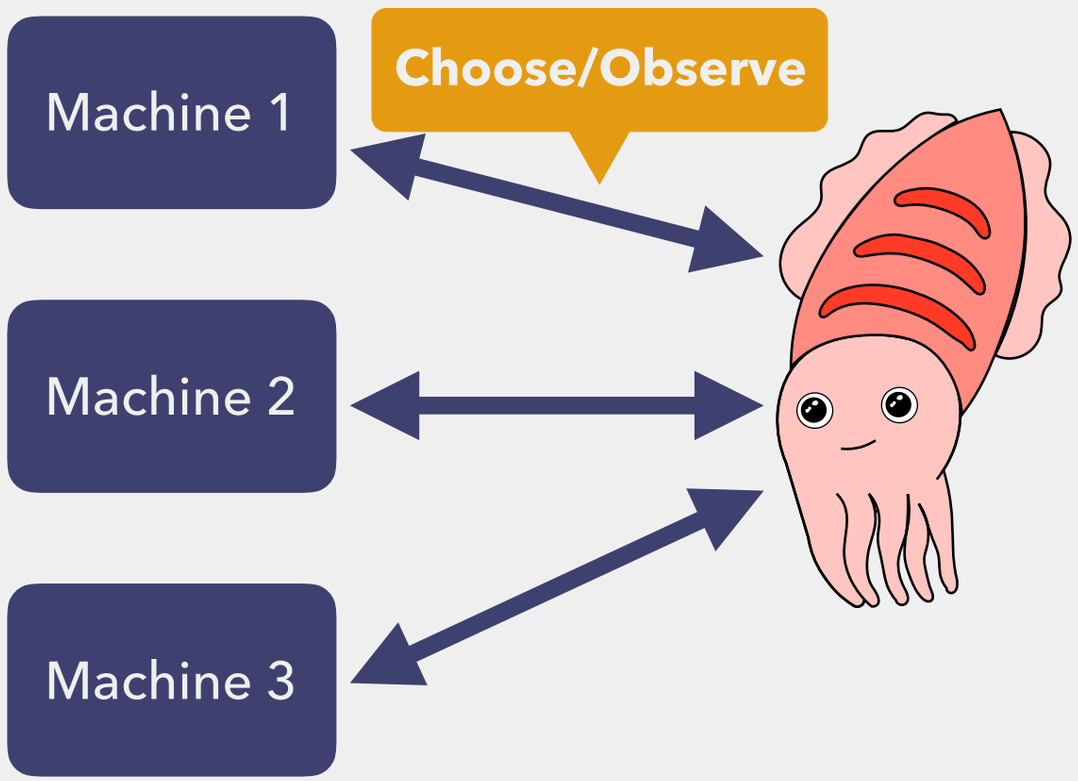
2. Synchronization & communication overheads

3. Feedback delay
    - How many times is `choose' called before an earlier reward is observed?
    - Fortunately, theoretically sound to have delays

Distributed Tuning Approach

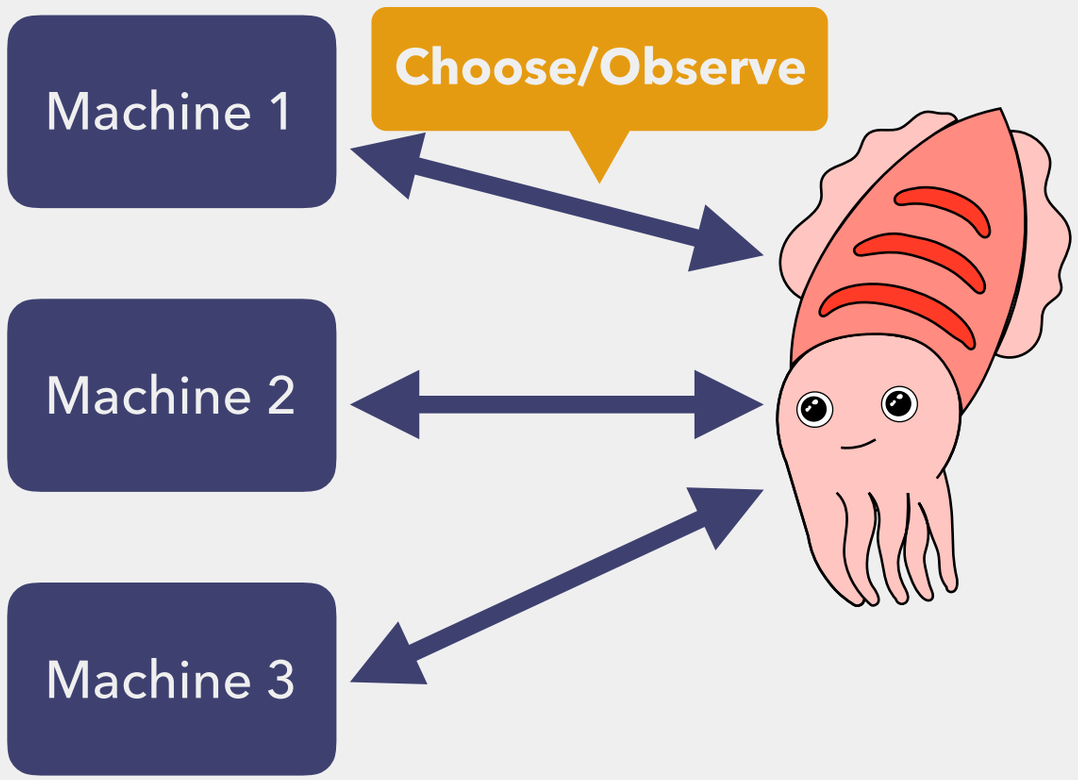
Distributed Tuning Approach

Centralized Tuner

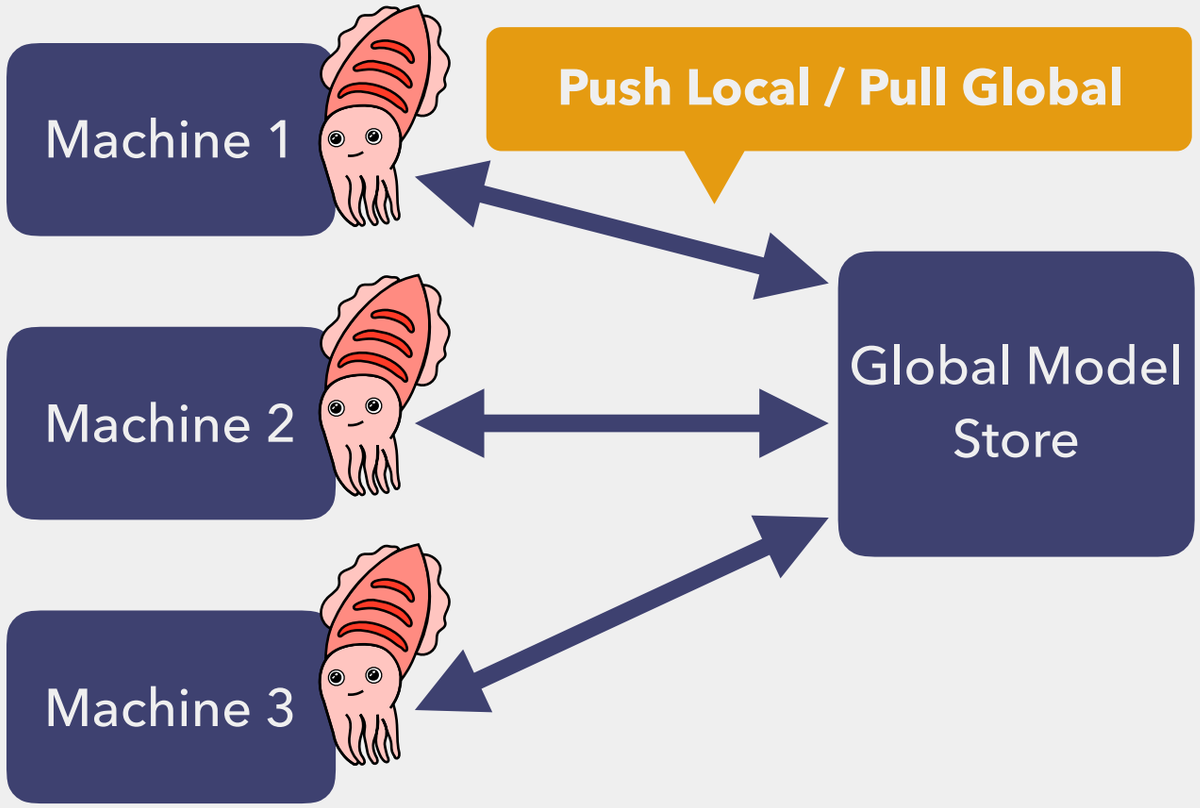


Distributed Tuning Approach

Centralized Tuner

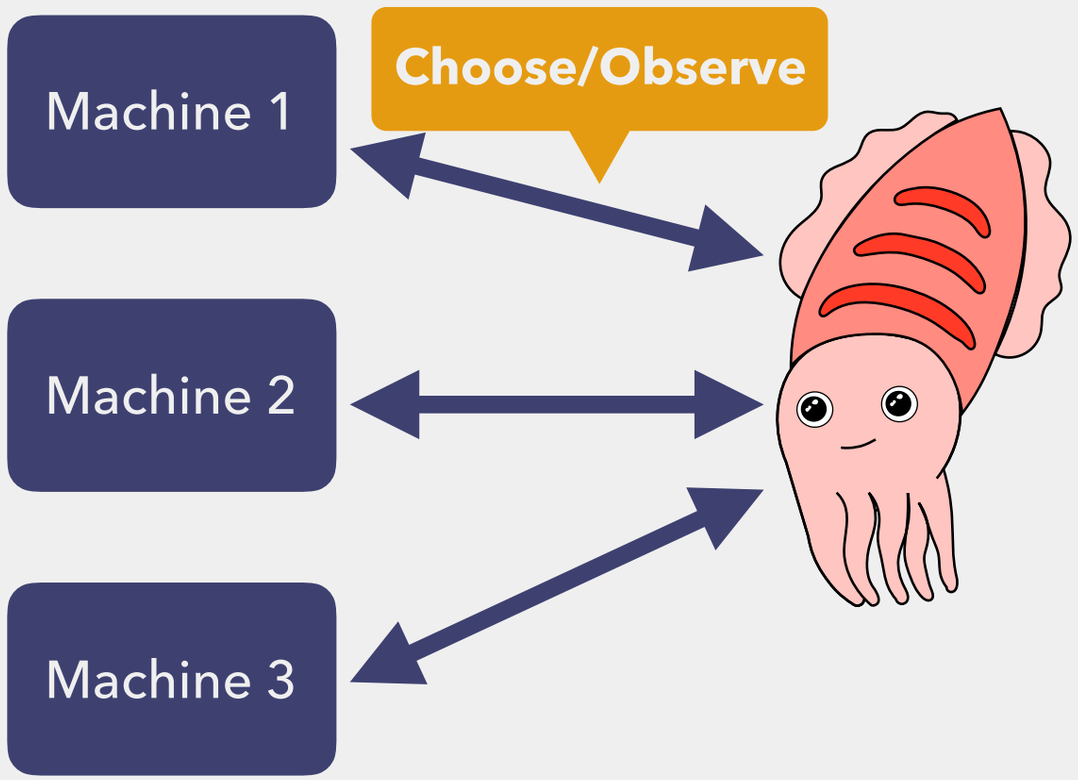


Independent Tuners, Centralized Store

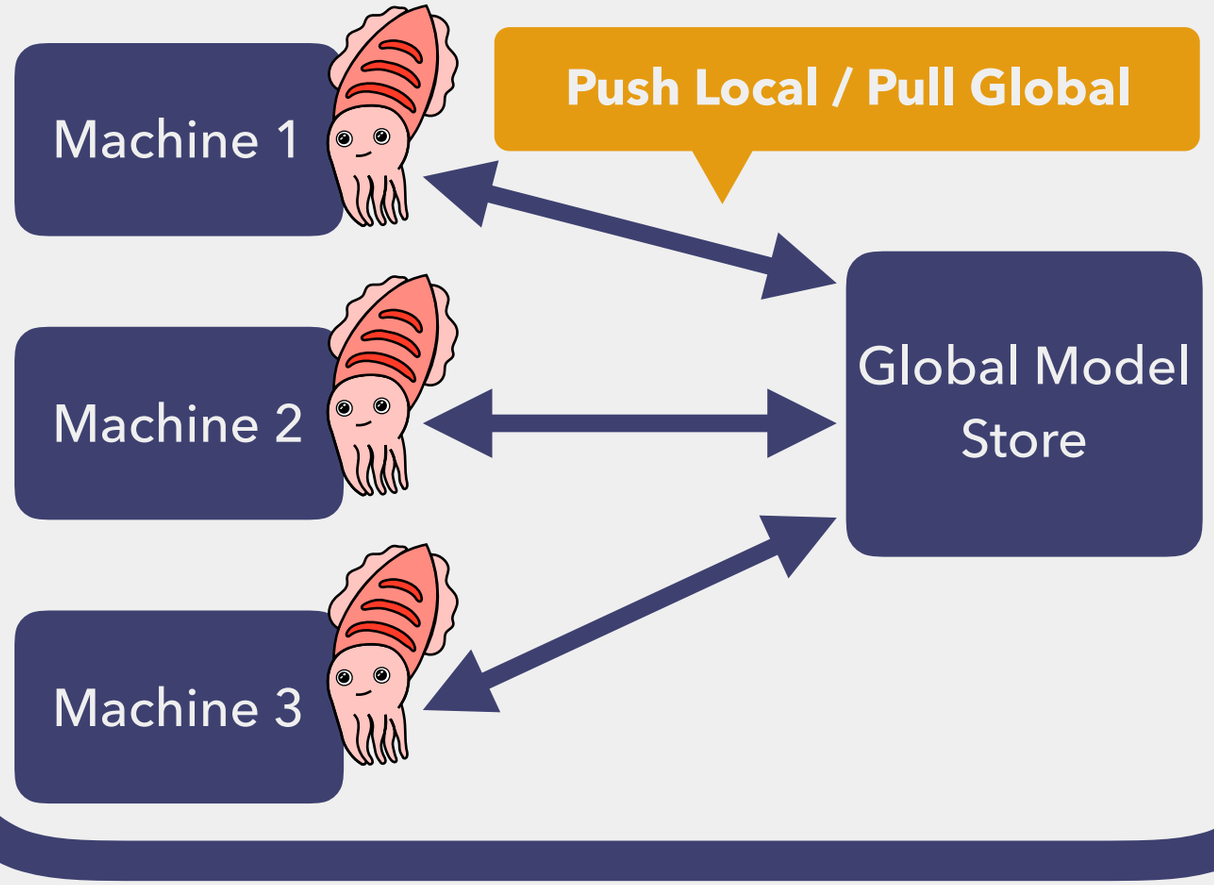


Distributed Tuning Approach

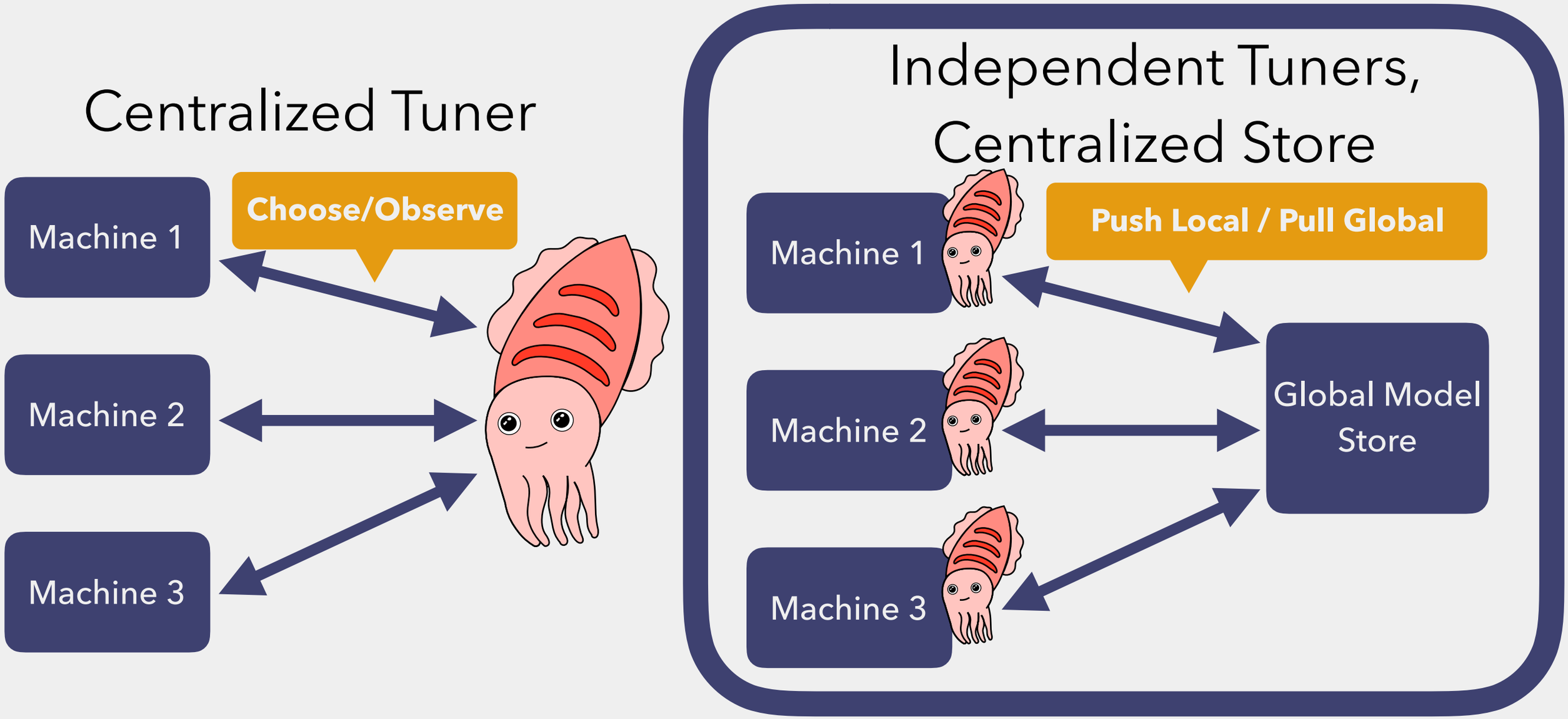
Centralized Tuner



Independent Tuners, Centralized Store

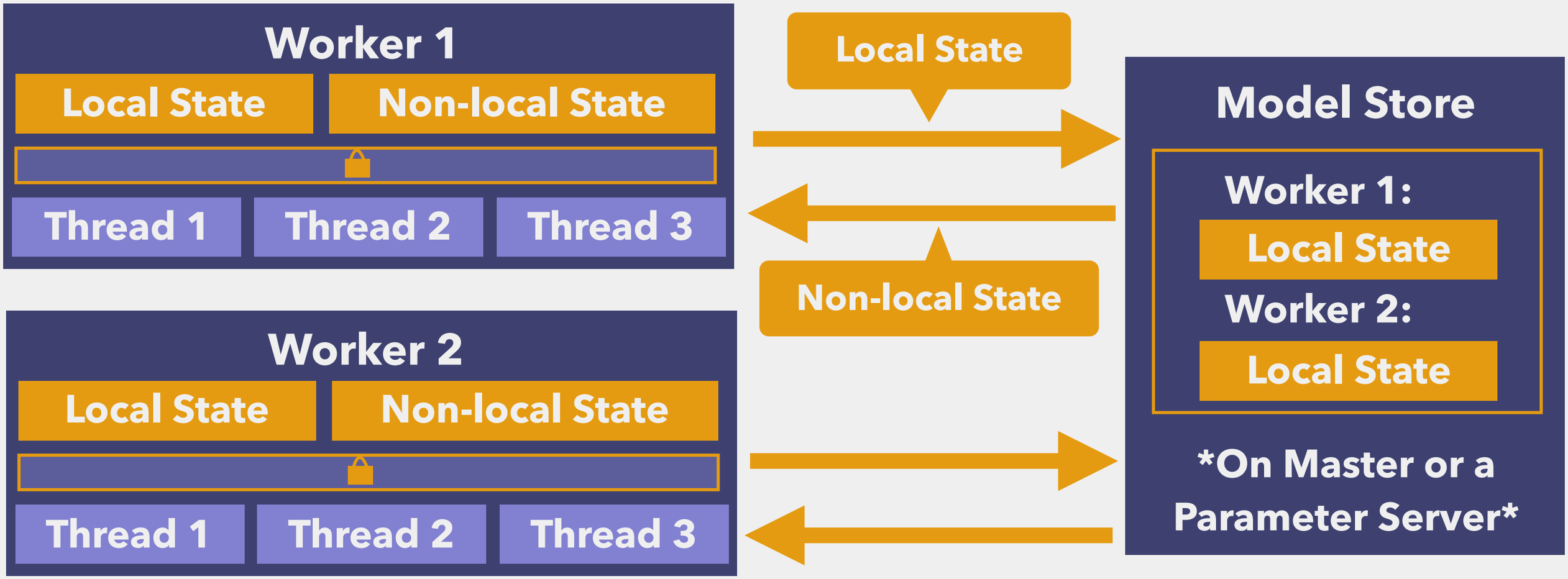


Distributed Tuning Approach

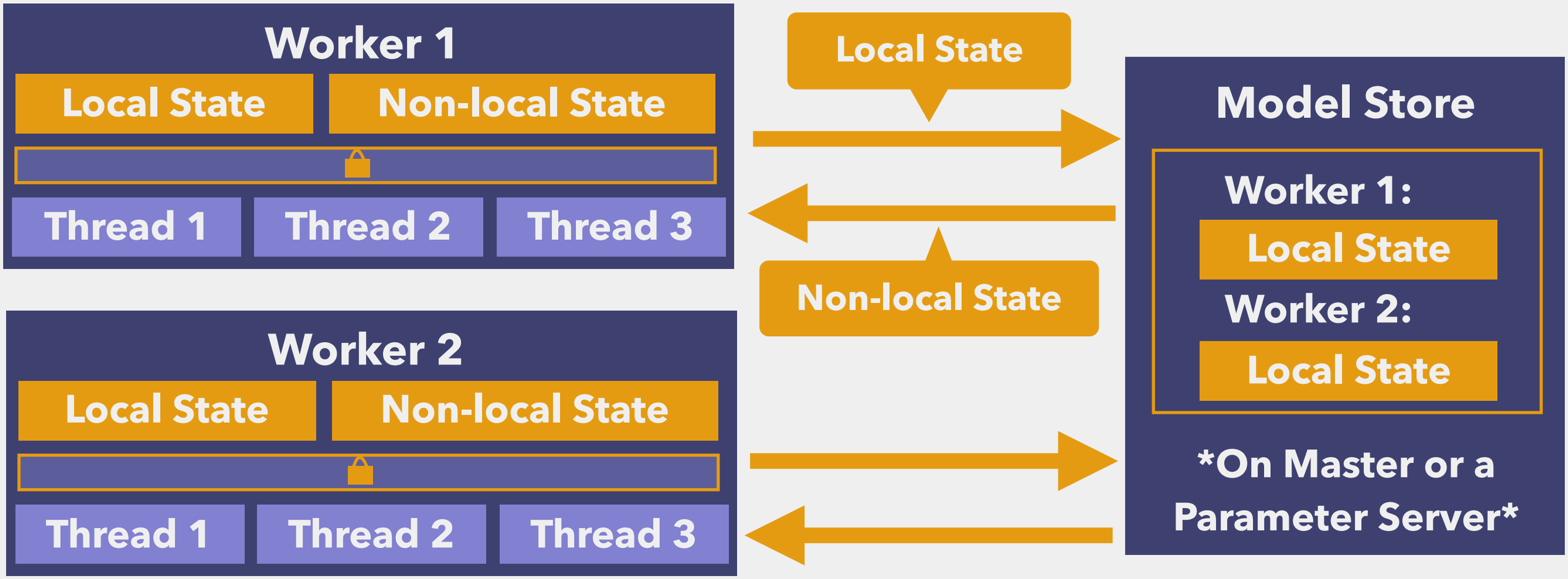


Peer-to-Peer is also a possibility,
but requires more communication

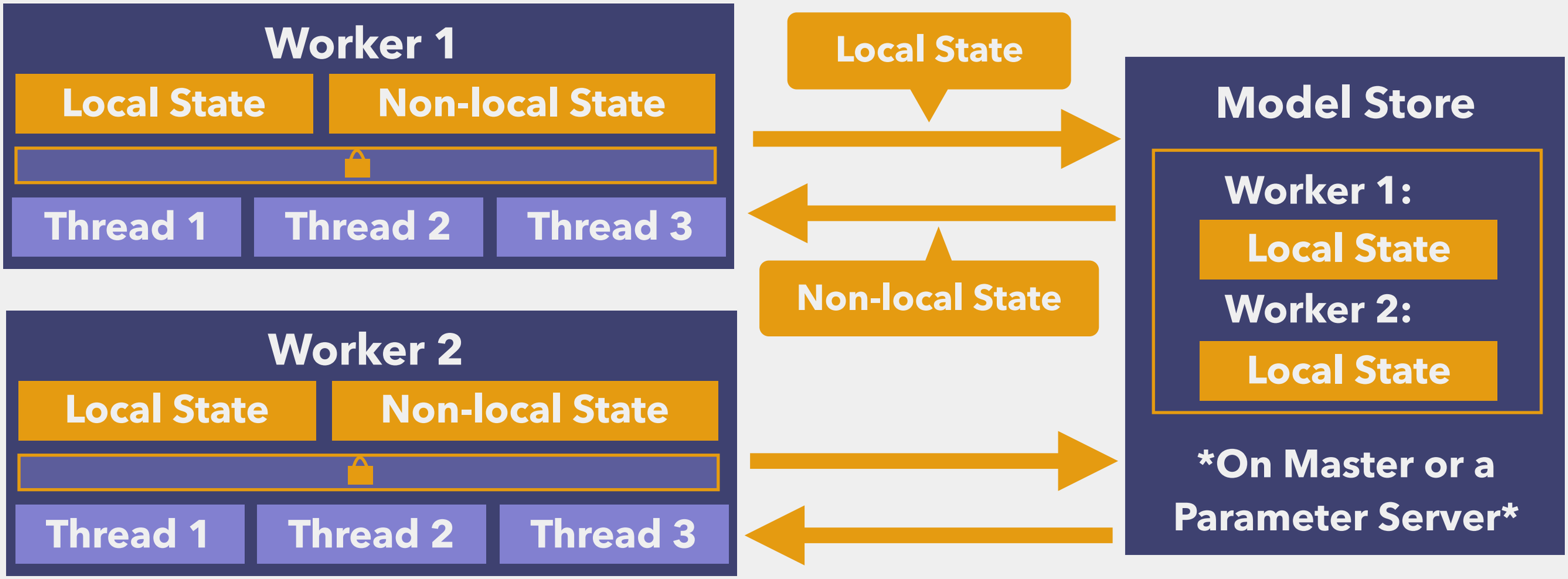
Distributed Tuning Approach



Distributed Tuning Approach

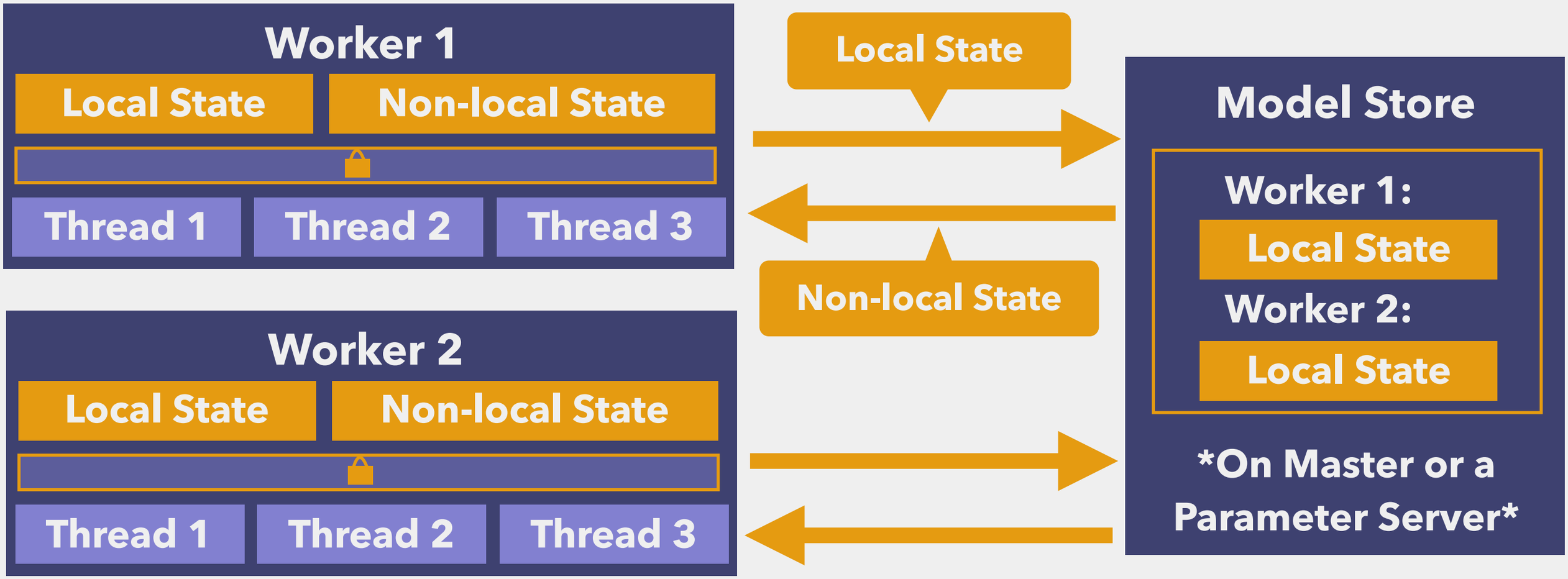


Distributed Tuning Approach



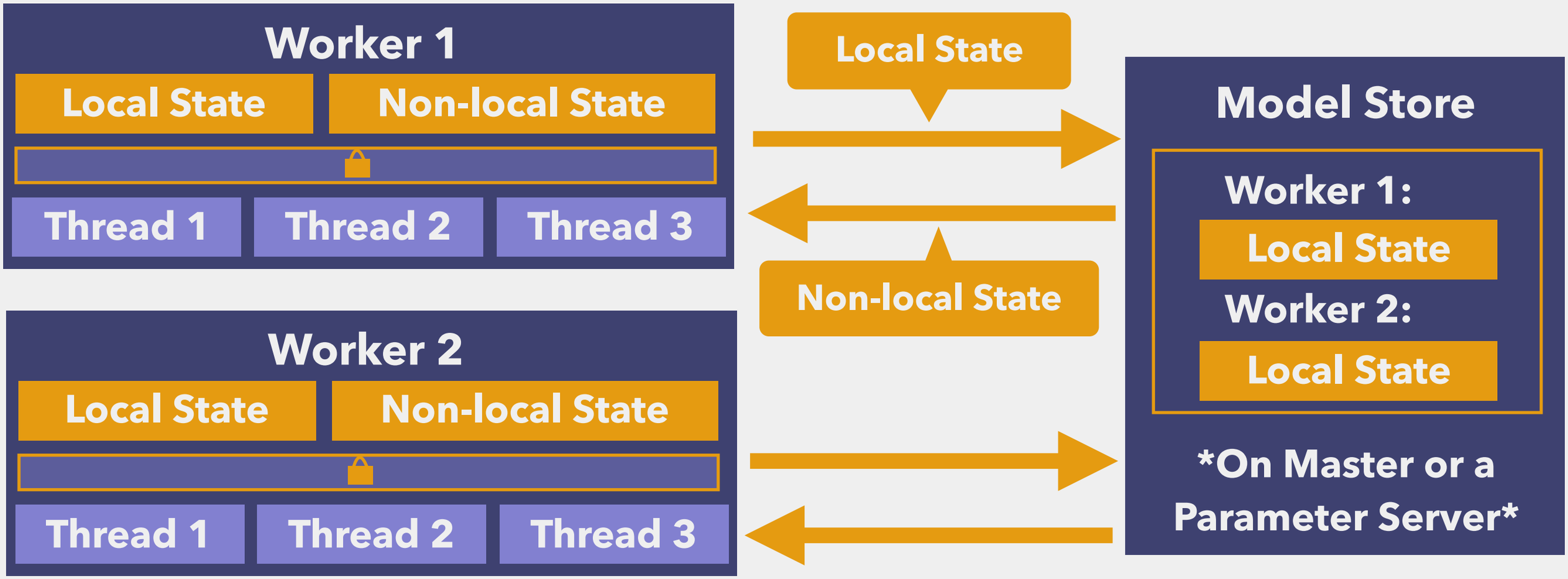
- When choosing: aggregate local & non-local state

Distributed Tuning Approach



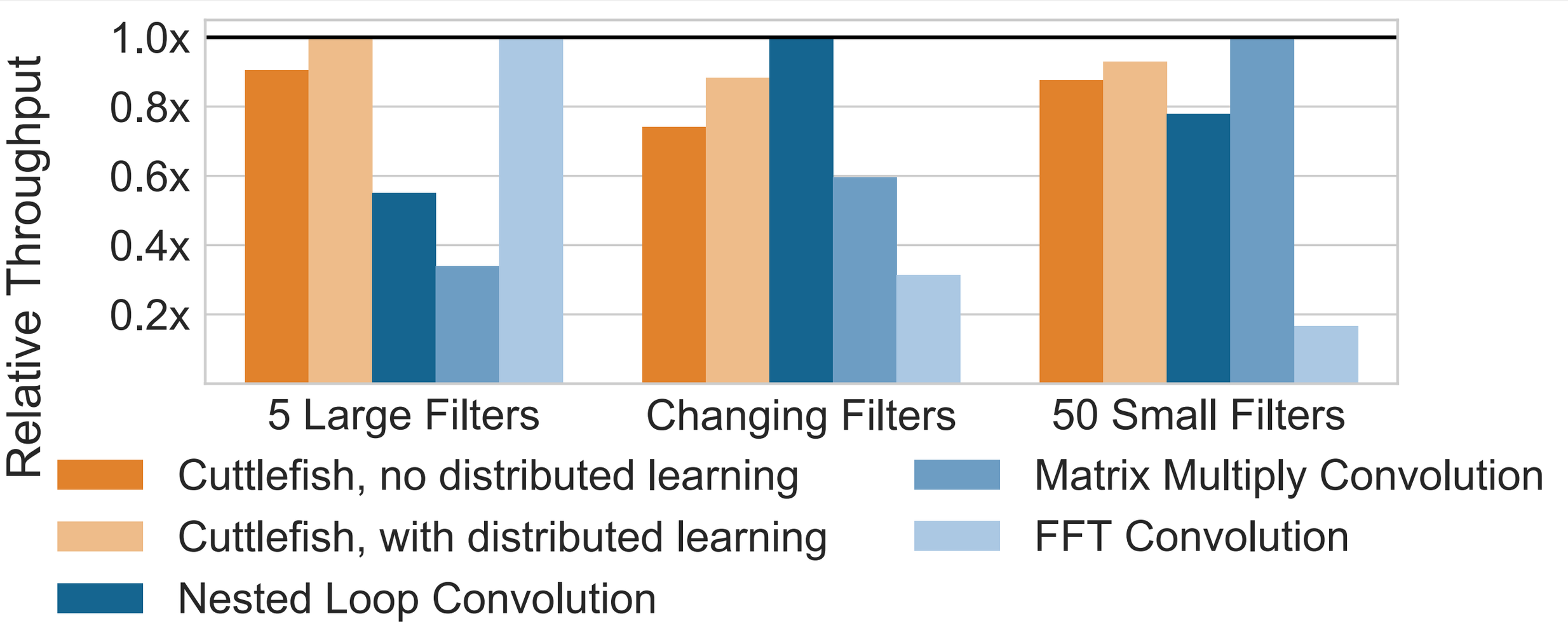
- When choosing: aggregate local & non-local state
- When observing: update the local state

Distributed Tuning Approach



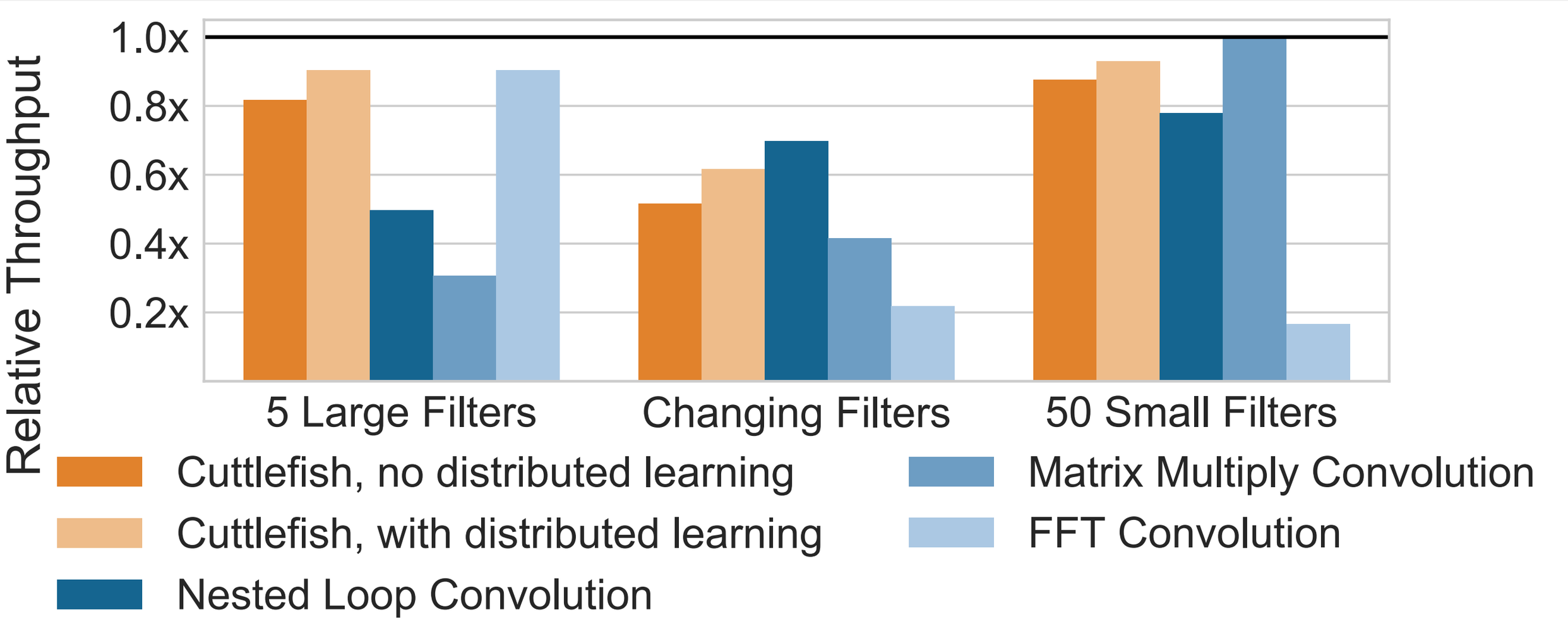
- When choosing: aggregate local & non-local state
- When observing: update the local state
- Model store aggregates non-local state

Results with Distributed Approach



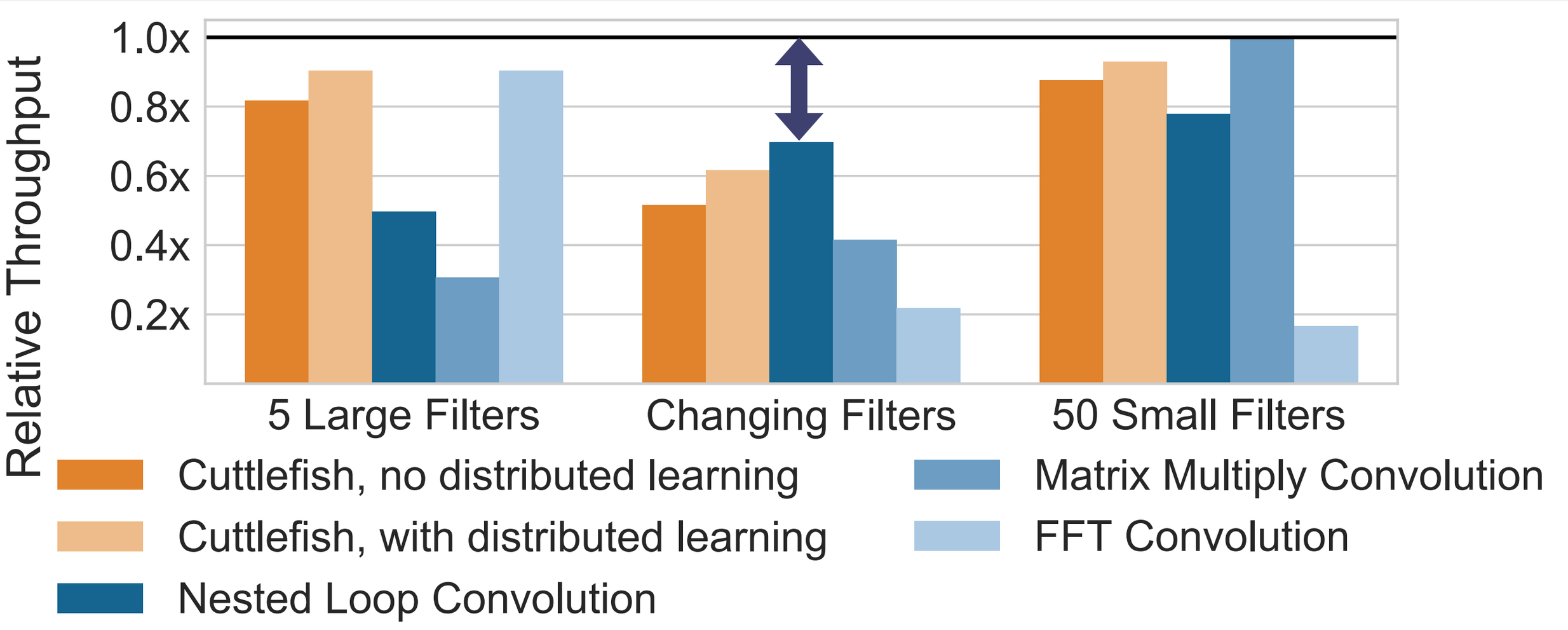
Relative throughput normalized against the highest-throughput algorithm

Results with Distributed Approach



Throughput normalized against an ideal oracle that always picks the fastest algorithm

Results with Distributed Approach



Throughput normalized against an ideal oracle that always picks the fastest algorithm

Cuttlefish

I. Problem & Motivation

II. The Cuttlefish API

III. Bandit-based Online Tuning

IV. Distributed Tuning Approach

V. Contextual Tuning (by learning cost models)

VI. Handling Nonstationary Settings

VII. Other Operators

VIII. Conclusion

Contextual Tuning

Contextual Tuning

- Best physical operator for each round may depend on current context
  - e.g. convolution performance depends on the image & filter dimensions

Contextual Tuning

- Best physical operator for each round may depend on current context
  - e.g. convolution performance depends on the image & filter dimensions
- Users may know important context features
  - e.g. from the asymptotic algorithmic complexity

Contextual Tuning

- Best physical operator for each round may depend on current context
  - e.g. convolution performance depends on the image & filter dimensions
- Users may know important context features
  - e.g. from the asymptotic algorithmic complexity
- Users can specify context in Tuner.choose

Contextual Tuning Algorithm

Contextual Tuning Algorithm

- Linear contextual Thompson sampling learns a linear model that maps features to rewards

Contextual Tuning Algorithm

- Linear contextual Thompson sampling learns a linear model that maps features to rewards
- Feature Normalization & Regularization
    - Increased robustness towards feature choices

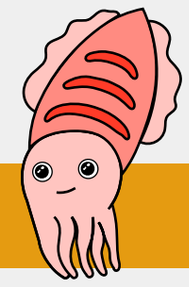
Contextual Tuning Algorithm

- Linear contextual Thompson sampling learns a linear model that maps features to rewards
- Feature Normalization & Regularization
    - Increased robustness towards feature choices
- Effectively learns a cost model

Tuning Convolution with Cuttlefish

```
def loopConvolve(image, filters): …
def fftConvolve(image, filters): …
def mmConvolve(image, filters): …

tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])
for image, filters in convolutions:
  convolve, token = tuner.choose()
  start = now()
  result = convolve(image, filters)
  elapsedTime = now() - start
  reward = computeReward(elapsedTime)
  tuner.observe(token, reward)
  output result
```

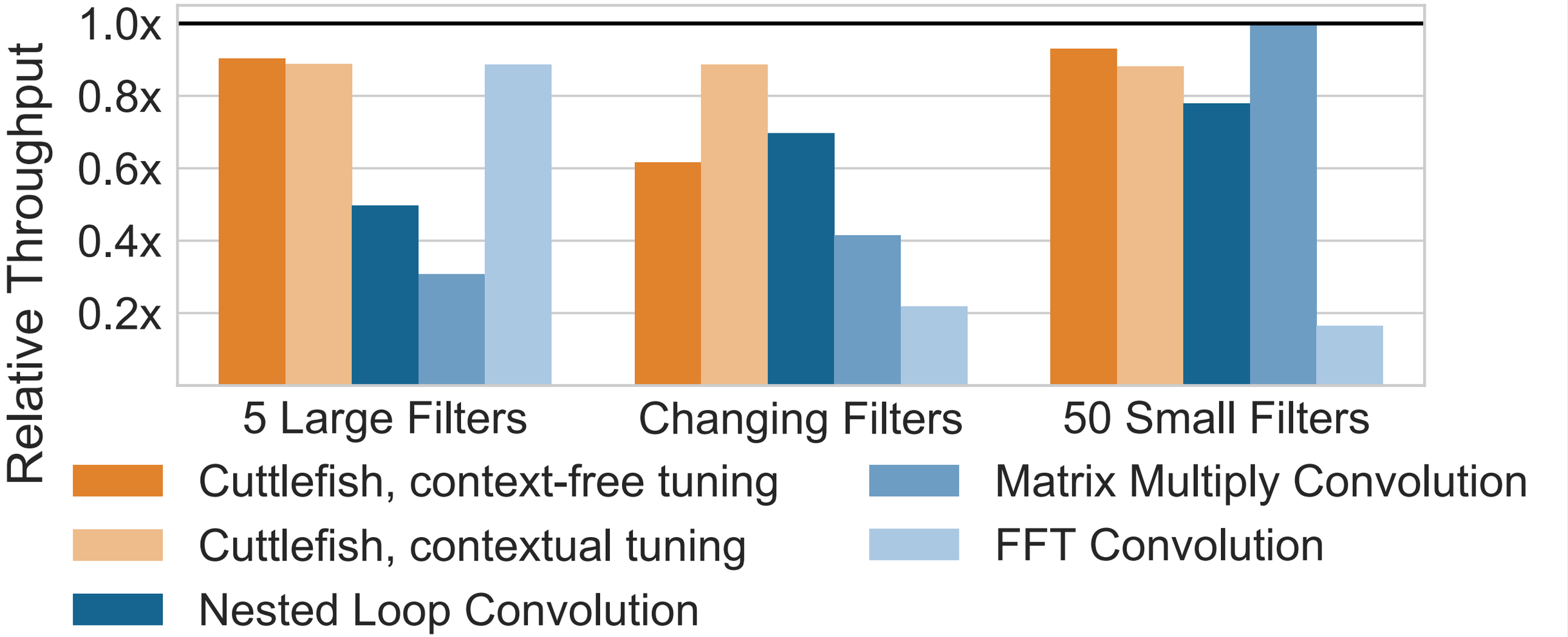


Tuning Convolution with Cuttlefish

```
def loopConvolve(image, filters): …
def fftConvolve(image, filters): …
def mmConvolve(image, filters): …
def getDimensions(image, filters): …
tuner = Tuner([loopConvolve, fftConvolve, mmConvolve])
```

```
for image, filters in convolutions:
  context = getDimensions(image, filters)
  convolve, token = tuner.choose(context)
  start = now()
  result = convolve(image, filters)
  elapsedTime = now() - start
  reward = computeReward(elapsedTime)
  tuner.observe(token, reward)
  output result
```

Contextual Convolution Results



Throughput normalized against an ideal oracle that always picks the fastest algorithm

Cuttlefish

I. Problem & Motivation

II. The Cuttlefish API

III. Bandit-based Online Tuning

IV. Distributed Tuning Approach

V. Contextual Tuning

VI. Handling Nonstationary Settings

VII. Other Operators

VIII. Conclusion

Nonstationary Settings

Nonstationary Settings

- Runtimes may drift over time, or differ across nodes
  - heterogenous cluster, changing resource availabilities, data properties varying throughout the workload, etc.
  - E.g. web crawl data and images may be stored sorted by website. This could correlate with performance

Nonstationary Settings

- Runtimes may drift over time, or differ across nodes
  - heterogenous cluster, changing resource availabilities, data properties varying throughout the workload, etc.
    - E.g. web crawl data and images may be stored sorted by website. This could correlate with performance

- We might not be capturing sufficient context!

Nonstationary Settings

- Runtimes may drift over time, or differ across nodes
  - heterogenous cluster, changing resource availabilities, data properties varying throughout the workload, etc.
    - E.g. web crawl data and images may be stored sorted by website. This could correlate with performance

- We might not be capturing sufficient context!

- Standard multi-armed bandit techniques fail

Nonstationary Settings

Nonstationary Settings

- Prior work: dynamic bandit approaches
  - Sliding windows, discounting older observations, reset on change detection, etc.
  - Good for dealing with changes over time

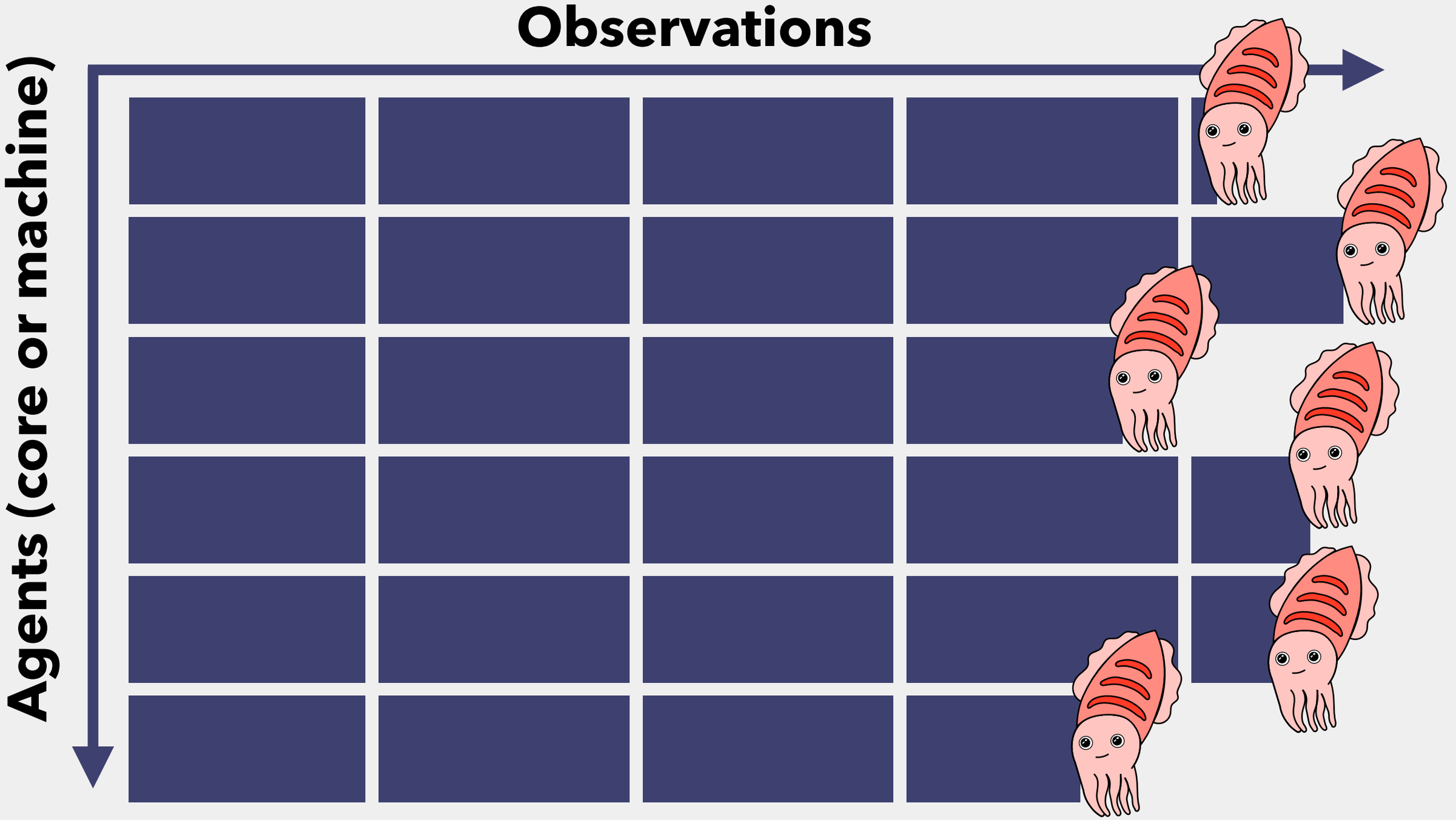
Nonstationary Settings

- Prior work: dynamic bandit approaches
  - Sliding windows, discounting older observations, reset on change detection, etc.
  - Good for dealing with changes over time
- Prior work: bandit clustering approaches
  - identify & share learning among agents solving similar bandit problems
  - Good for dealing with differences between cores

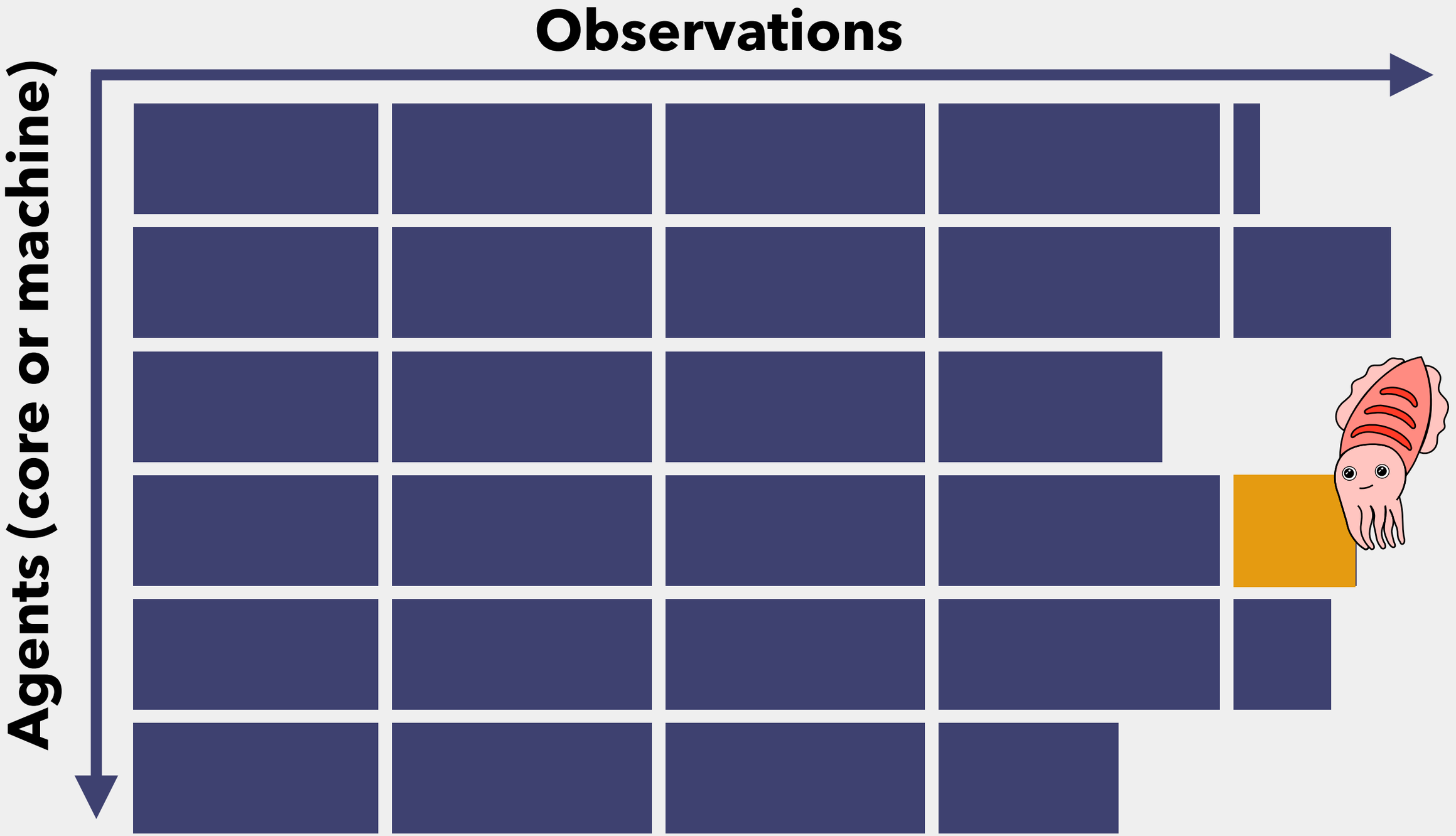
Nonstationary Settings

- Prior work: dynamic bandit approaches
  - Sliding windows, discounting older observations, reset on change detection, etc.
  - Good for dealing with changes over time
- Prior work: bandit clustering approaches
  - identify & share learning among agents solving similar bandit problems
  - Good for dealing with differences between cores
- Need dynamic bandit clustering where agents' underlying problems may change over time!

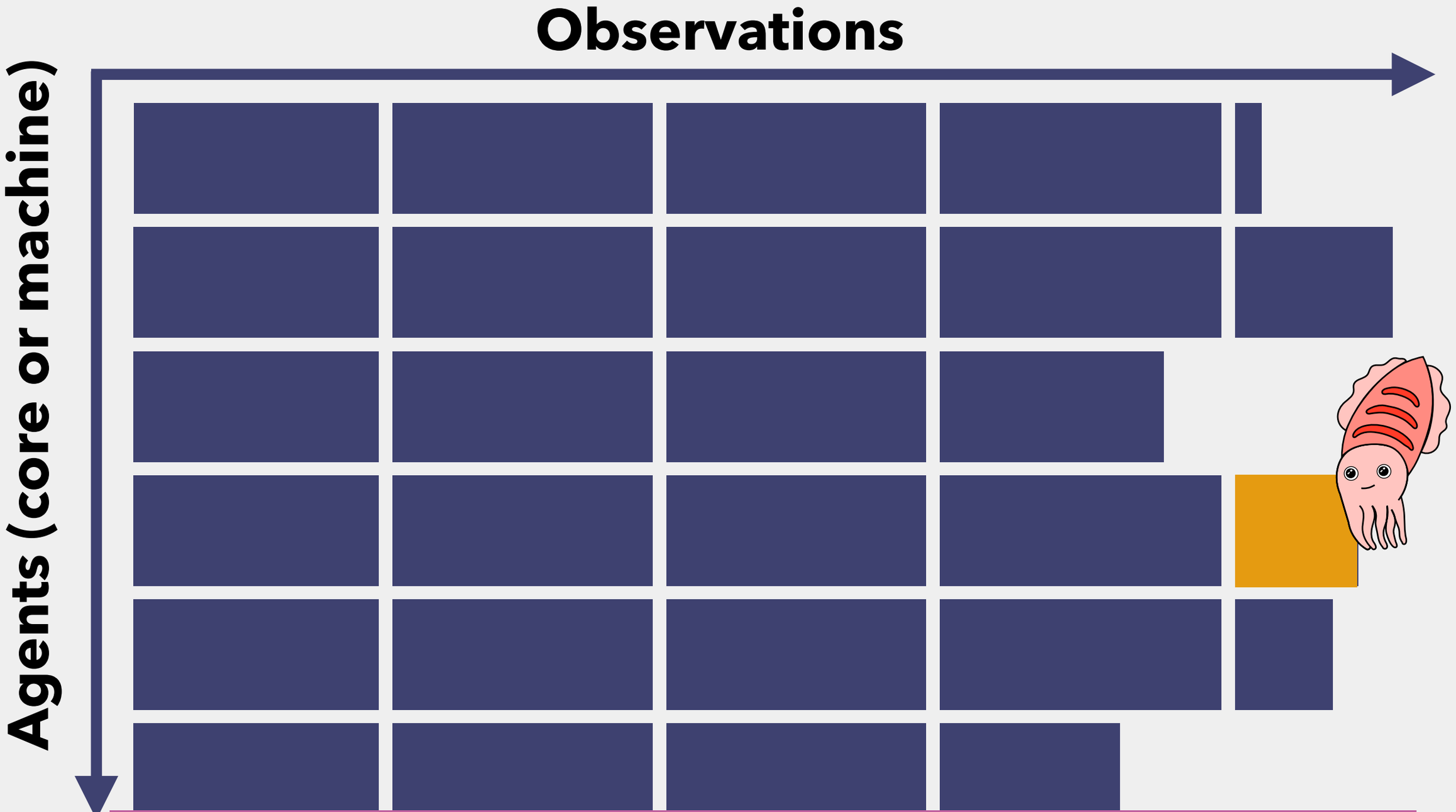
Possible Solution



Possible Solution

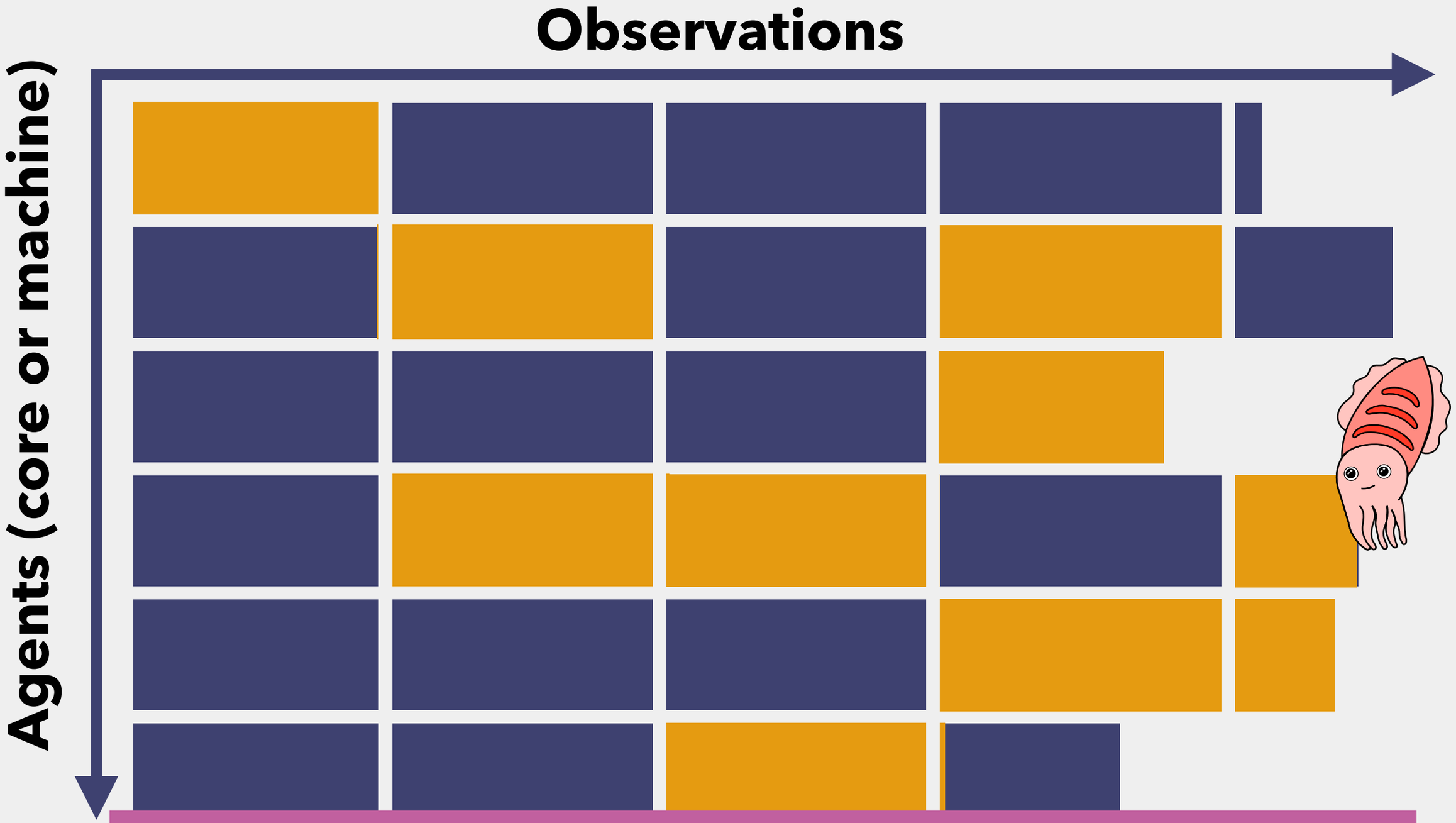


Possible Solution



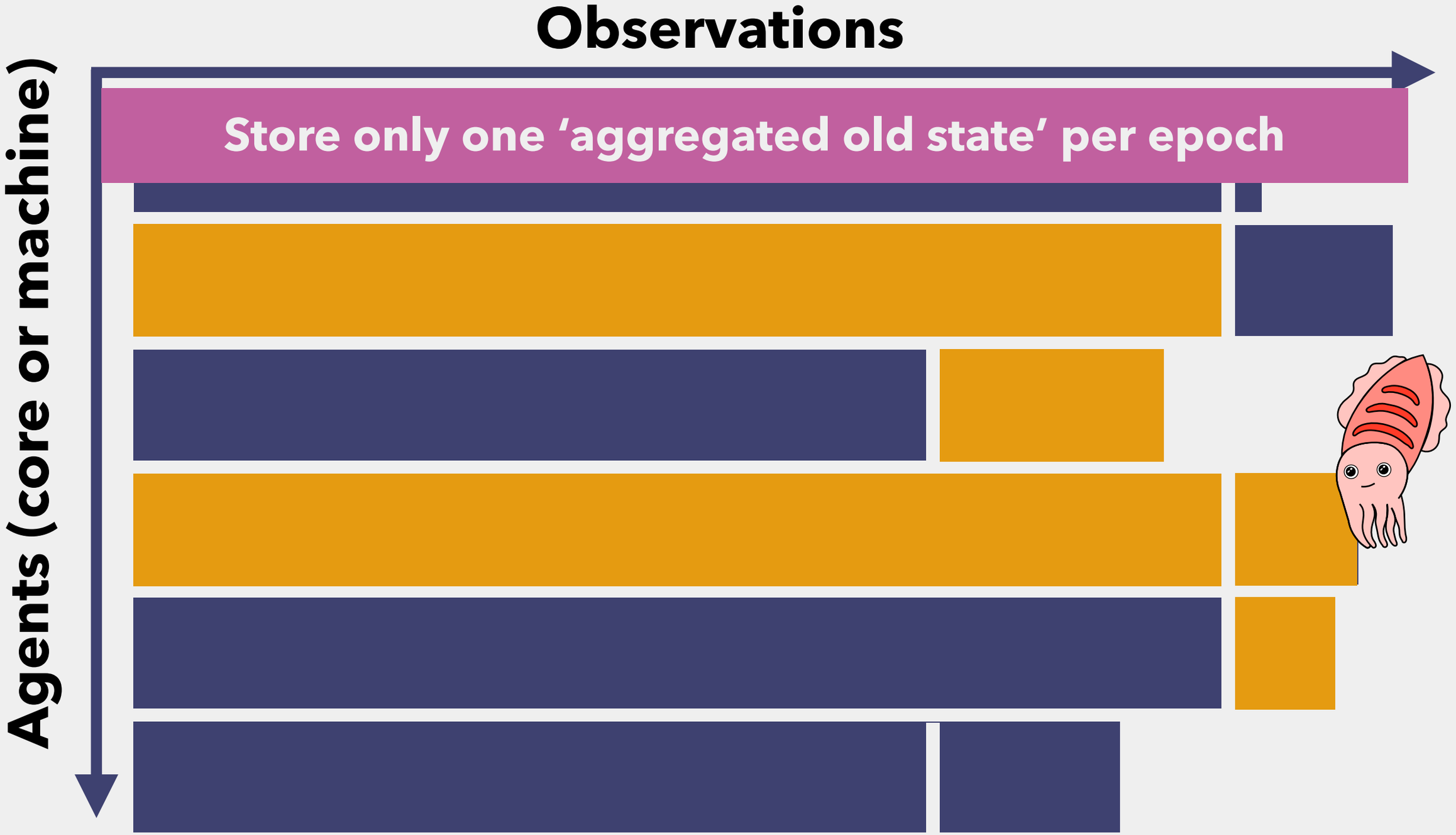
Use all epochs that pass a statistical similarity test

Possible Solution

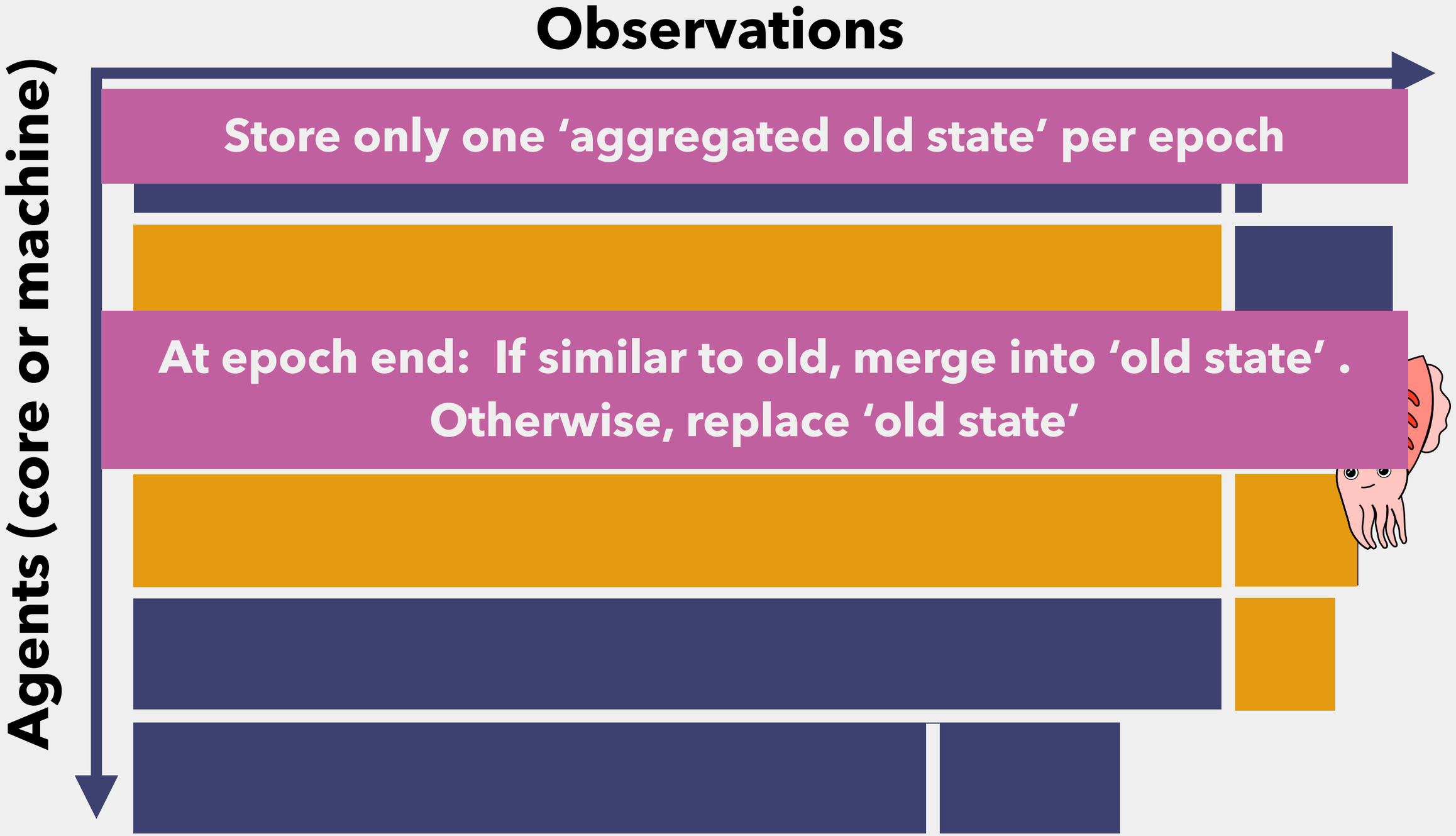


Use all epochs that pass a statistical similarity test

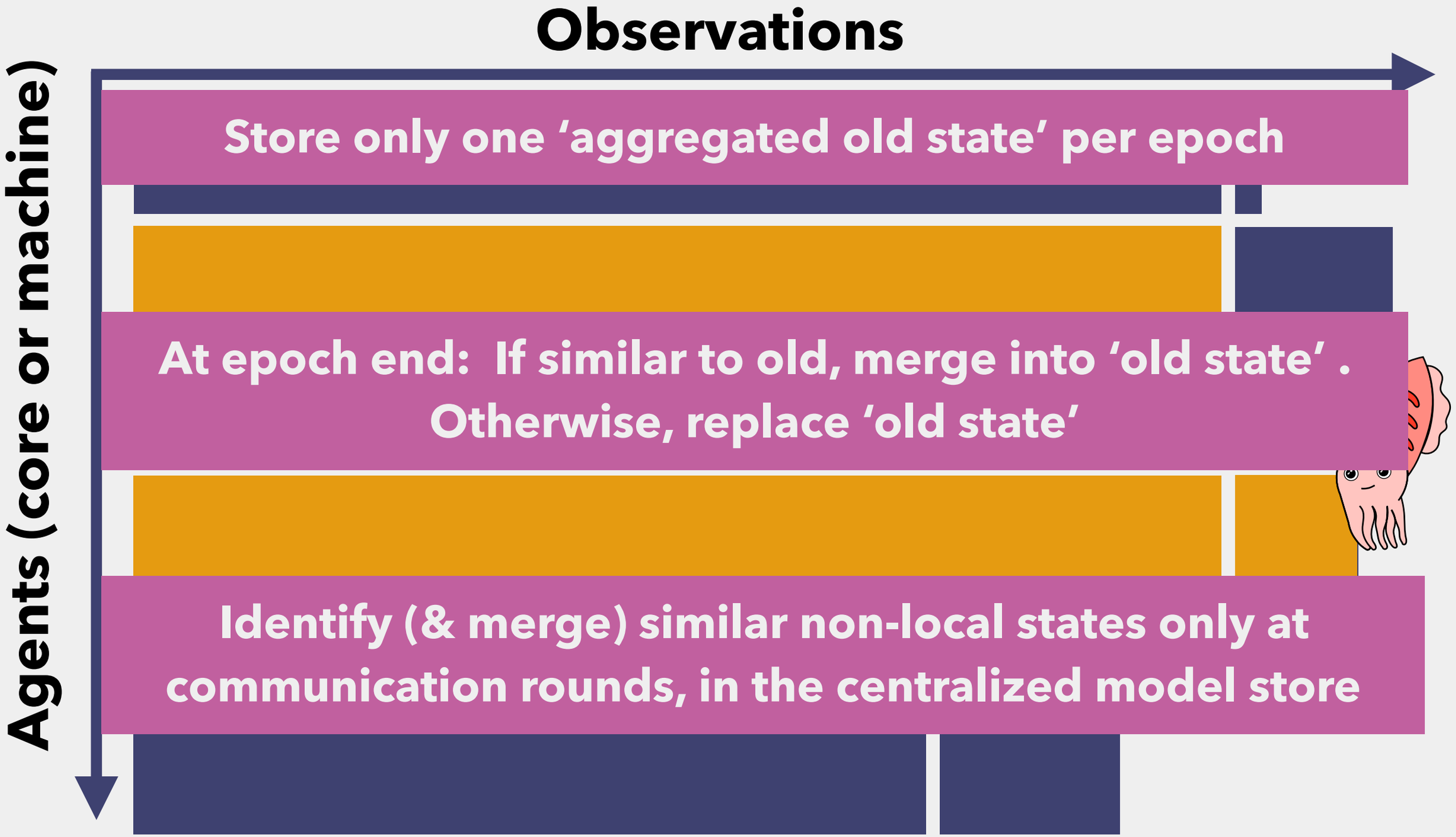
To Lower Overheads



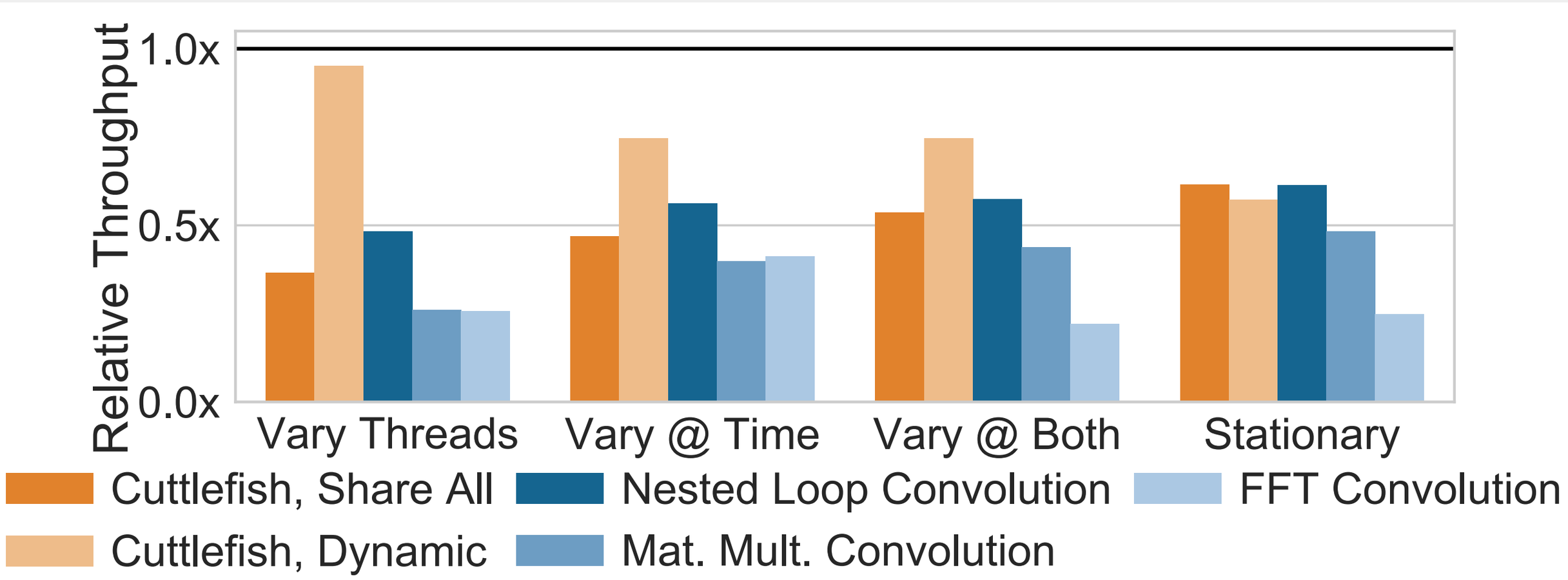
To Lower Overheads



To Lower Overheads



Nonstationary Results



Throughput normalized against an ideal oracle that always picks the fastest algorithm

Cuttlefish

I. Problem & Motivation

II. The Cuttlefish API

III. Bandit-based Online Tuning

IV. Distributed Tuning Approach

V. Contextual Tuning

VI. Handling Nonstationary Settings

VII. Other Operators

VIII. Conclusion

Regex Operator

Regex Operator

- Tune between four regular expression searching libraries
    - Built-in Java Regex and 3 third-party libraries

Regex Operator

- Tune between four regular expression searching libraries
    - Built-in Java Regex and 3 third-party libraries
- Search through 256k Common Crawl docs (~30gb uncompressed)
    - one tuning round per doc

Regex Operator

- Tune between four regular expression searching libraries
    - Built-in Java Regex and 3 third-party libraries
- Search through 256k Common Crawl docs (~30gb uncompressed)
    - one tuning round per doc
- Test 8 Regexes sourced from regex-sharing website RegExr
    - Match hyperlinks, trigrams, valid emails, color codes, etc.

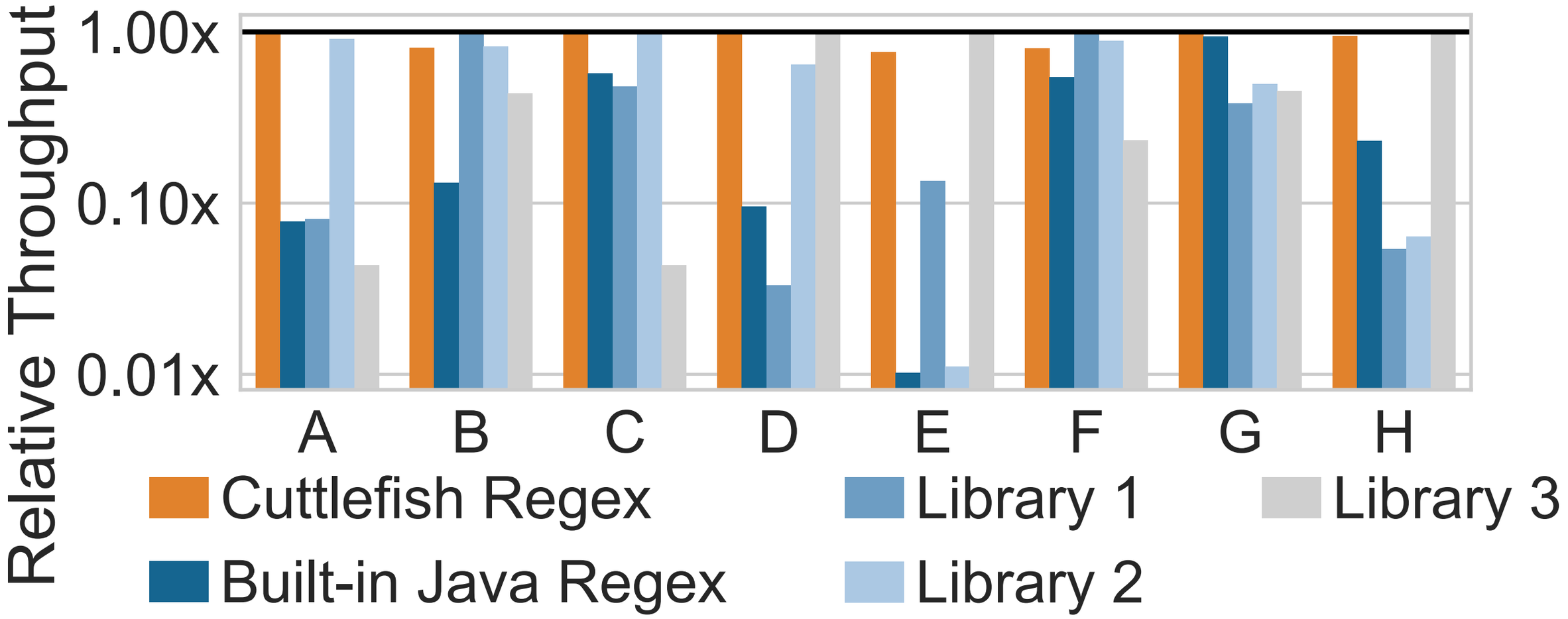
Regex Operator

- Tune between four regular expression searching libraries
    - Built-in Java Regex and 3 third-party libraries
- Search through 256k Common Crawl docs (~30gb uncompressed)
    - one tuning round per doc
- Test 8 Regexes sourced from regex-sharing website RegExr
    - Match hyperlinks, trigrams, valid emails, color codes, etc.
- Multiple of orders of magnitude variation in performance
    - Email validation regex w/ built-in java utilities takes 33µs to process the fastest document, but over 1000s for the slowest document

Regex Operator

- Tune between four regular expression searching libraries
    - Built-in Java Regex and 3 third-party libraries
- Search through 256k Common Crawl docs (~30gb uncompressed)
    - one tuning round per doc
- Test 8 Regexes sourced from regex-sharing website RegExr
    - Match hyperlinks, trigrams, valid emails, color codes, etc.
- Multiple of orders of magnitude variation in performance
    - Email validation regex w/ built-in java utilities takes 33μs to process the fastest document, but over 1000s for the slowest document
- 8-node (AWS EC2 4-core r3.xlarge) cluster

Regex Results



Note: Y-axis is Log-scale

Distributed Parallel Join Operator

Distributed Parallel Join Operator

- Hash-partition relations according to join attributes

Distributed Parallel Join Operator

- Hash-partition relations according to join attributes
- On each partition, pick a local hash join or a local sort-merge join

Distributed Parallel Join Operator

- Hash-partition relations according to join attributes
- On each partition, pick a local hash join or a local sort-merge join
- Rewards capture total join time
  - measure from when joins begin until result iterators are fully consumed

Distributed Parallel Join Operator

- Hash-partition relations according to join attributes
- On each partition, pick a local hash join or a local sort-merge join
- Rewards capture total join time
  - measure from when joins begin until result iterators are fully consumed
- Set as Spark SQL 2.2's join for all equijoins too large to broadcast
  - No heuristics and cost models in the query optimizer, falls back on explicit configurations (defaults to global sort-merge join)

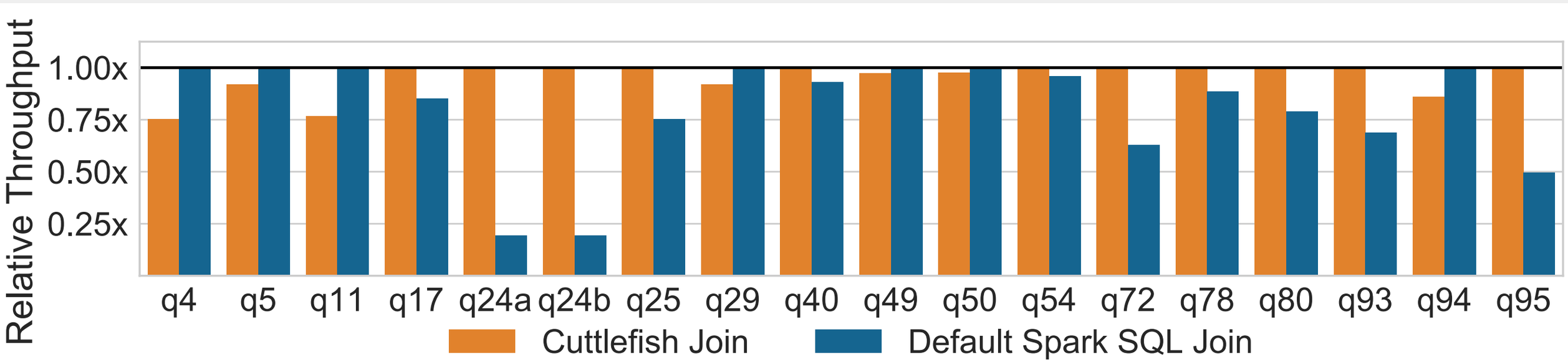
Distributed Parallel Join Operator

- Hash-partition relations according to join attributes
- On each partition, pick a local hash join or a local sort-merge join
- Rewards capture total join time
  - measure from when joins begin until result iterators are fully consumed
- Set as Spark SQL 2.2's join for all equijoins too large to broadcast
  - No heuristics and cost models in the query optimizer, falls back on explicit configurations (defaults to global sort-merge join)
- Test on TPC-DS benchmark (scale factor 200)

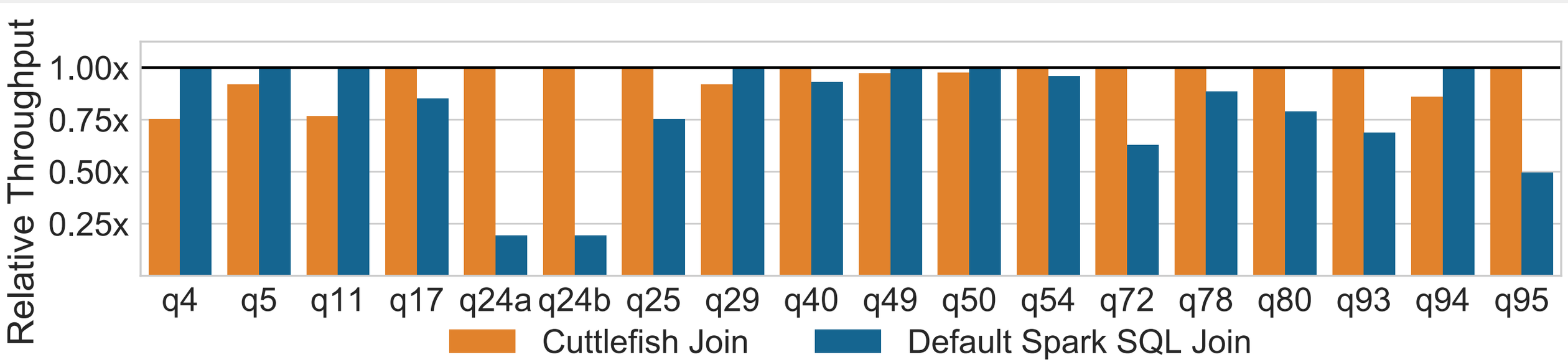
Distributed Parallel Join Operator

- Hash-partition relations according to join attributes
- On each partition, pick a local hash join or a local sort-merge join
- Rewards capture total join time
  - measure from when joins begin until result iterators are fully consumed
- Set as Spark SQL 2.2's join for all equijoins too large to broadcast
  - No heuristics and cost models in the query optimizer, falls back on explicit configurations (defaults to global sort-merge join)
- Test on TPC-DS benchmark (scale factor 200)
- Configure queries to use 512 shuffle / join partitions

Join Results (Query Throughput)

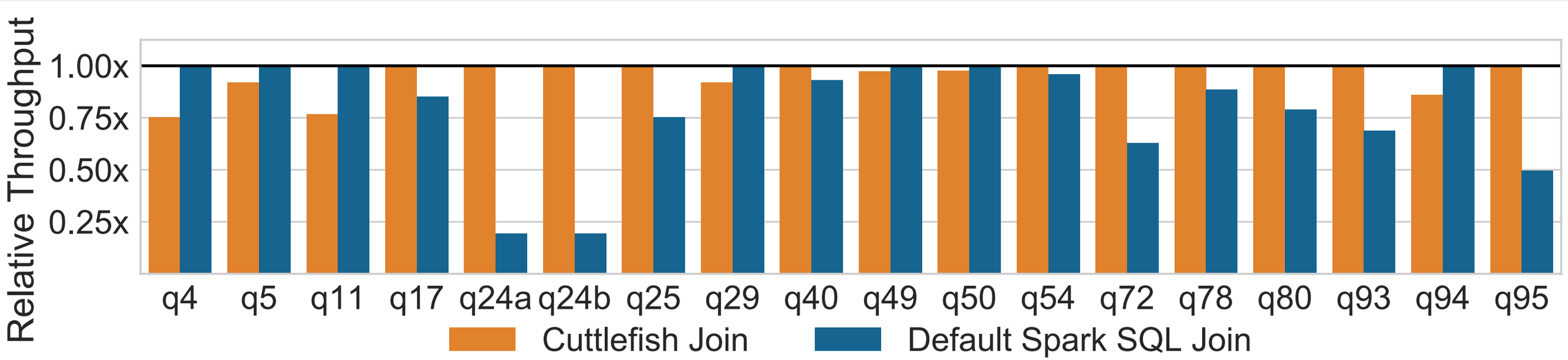


Join Results (Query Throughput)



But, requires exploration &
provides no ‘special ordering’ benefits

Join Results (Query Throughput)



Cuttlefish join usually faster

(Join throughput graphs even more dramatic)

But, requires exploration &

provides no ‘special ordering’ benefits

Cuttlefish

I. Problem & Motivation

II. The Cuttlefish API

III. Bandit-based Online Tuning

IV. Distributed Tuning Approach

V. Contextual Tuning

VI. Handling Nonstationary Settings

VII. Other Operators

VIII. Conclusion

Cuttlefish

- A simple, flexible API for online tuning
- Thompson-sampling based tuning algorithms
- Supports contextual tuning (learns cost models)
- Distributed learning between workers
- Adapts to nonstationary workloads
- Prototyped in Apache Spark & successfully tunes convolution, regex, and join operators