Realizing value in shared compute infrastructures

Andrew Chung

Thesis Committee:
Greg Ganger (Chair)
Phil Gibbons
George Amvrosiadis
Carlo Curino (Microsoft GSL)

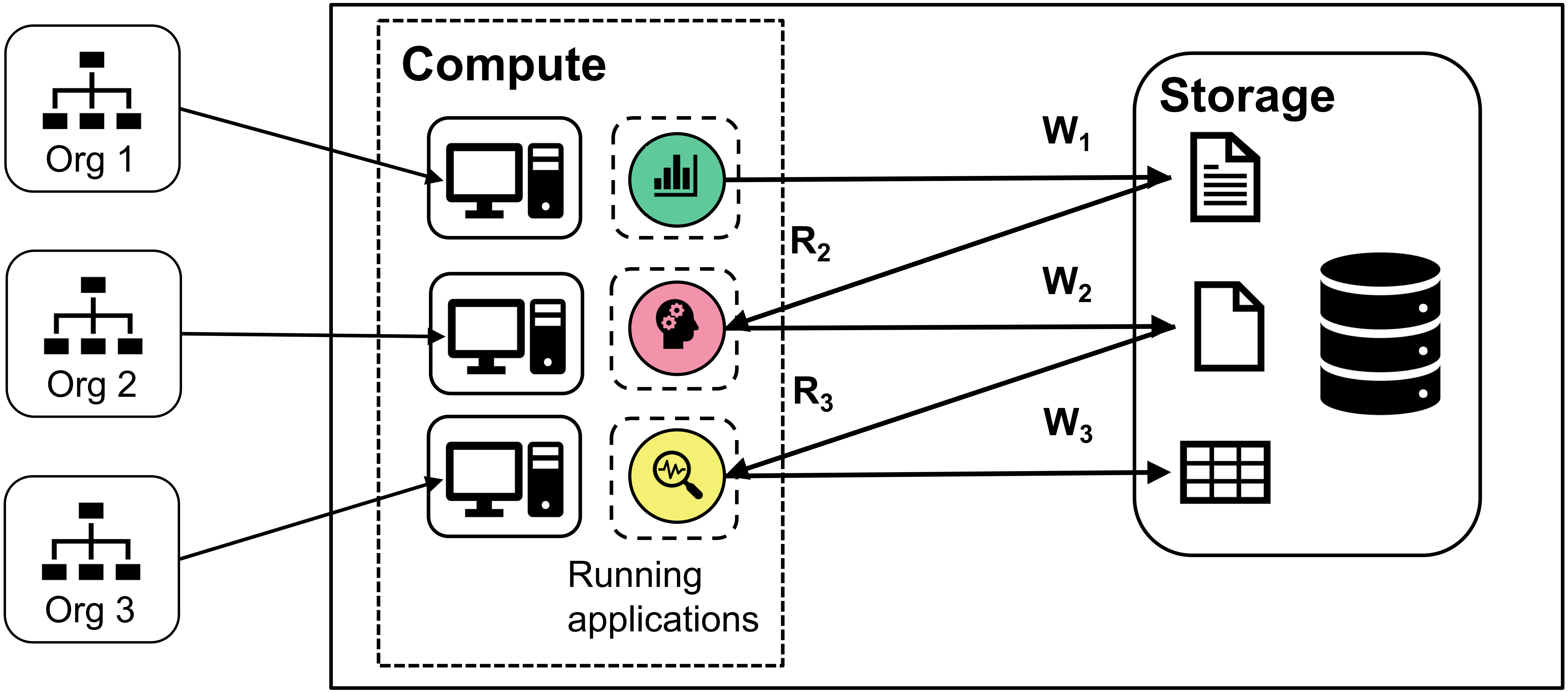
Talk outline

- Shared cluster environments
- Thesis statement
- Prior work: Realizing value through user applications
- Ongoing work: Realizing value through dependency-aware resource management
- Thesis timeline

Shared cluster environments

- Highly heterogeneous resources and applications
  - Many users from various groups and organizations
- Increasingly-common:
  - Data shared across users, groups, and organizations
- Examples of shared cluster environments:
  - Public clouds (AWS, Azure, GCE)
  - Private clouds (Microsoft's Cosmos clusters)

Example: Shared cluster environment



User goals

- **Users:** Run applications in shared environment
  - Goal 1: Meet application business requirements
  - Goal 2: Cost-effective resource acquisition
- Challenges:
  - Resource heterogeneity
  - Wide variety of pricing mechanisms

Cluster operator goals

- **Cluster operators:** Maximize profit & satisfy users
  - Goal 1: Prioritize resource allocation to applications
    - Using some notion of “user value”
  - Goal 2: Efficiently manage cluster operation costs
- Challenges:
  - Resource heterogeneity and availability
  - Hidden user values and performance requirements

Thesis statement

Value-realized in shared data environments can be improved both by value- and dependency-aware resource management systems from cluster operators and by cost- and heterogeneity-aware applications from users.

Realizing value through user application frameworks

Background: Public clouds

- Public clouds offer a variety of resources
  - e.g., varying compute capacity, storage, HW accelerators
- Under different types of contracts
  - e.g., reliable, transient, and burst
- Difficult for users to choose resources cost-effectively!

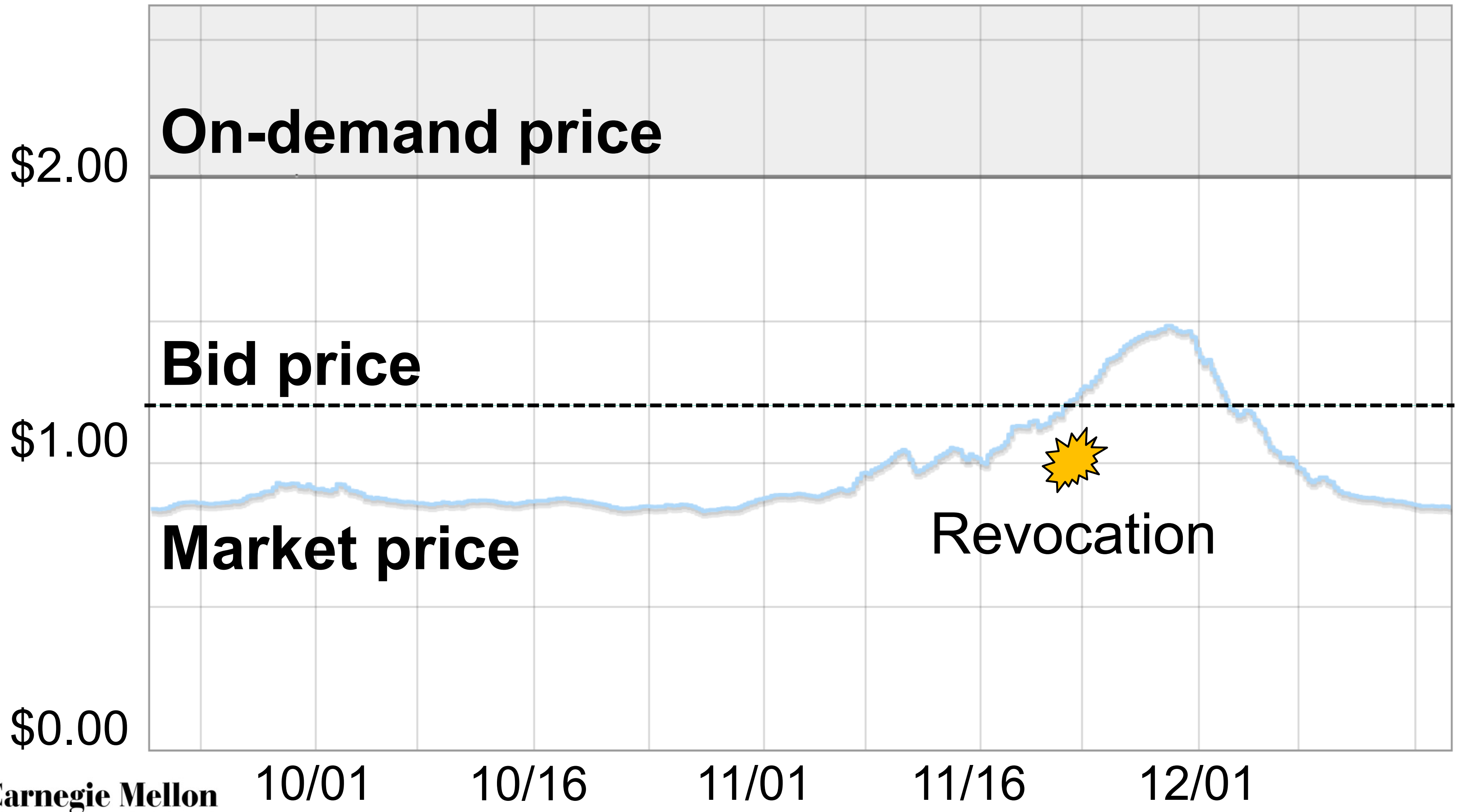
Achieve user value through:
Application-specific, cost-aware resource acquisition

Application-specific resource acquisition: Case studies

1. Elastic web services
    - Spot-dancing for elastic services with latency SLOs
    - Tributary [USENIX ATC 2018]
2. General containerized batch task scheduling
    - Cost-aware container scheduling in the public cloud
    - Stratus [ACM SoCC 2018]
        - Best student paper award

Transient/spot instances in AWS EC2

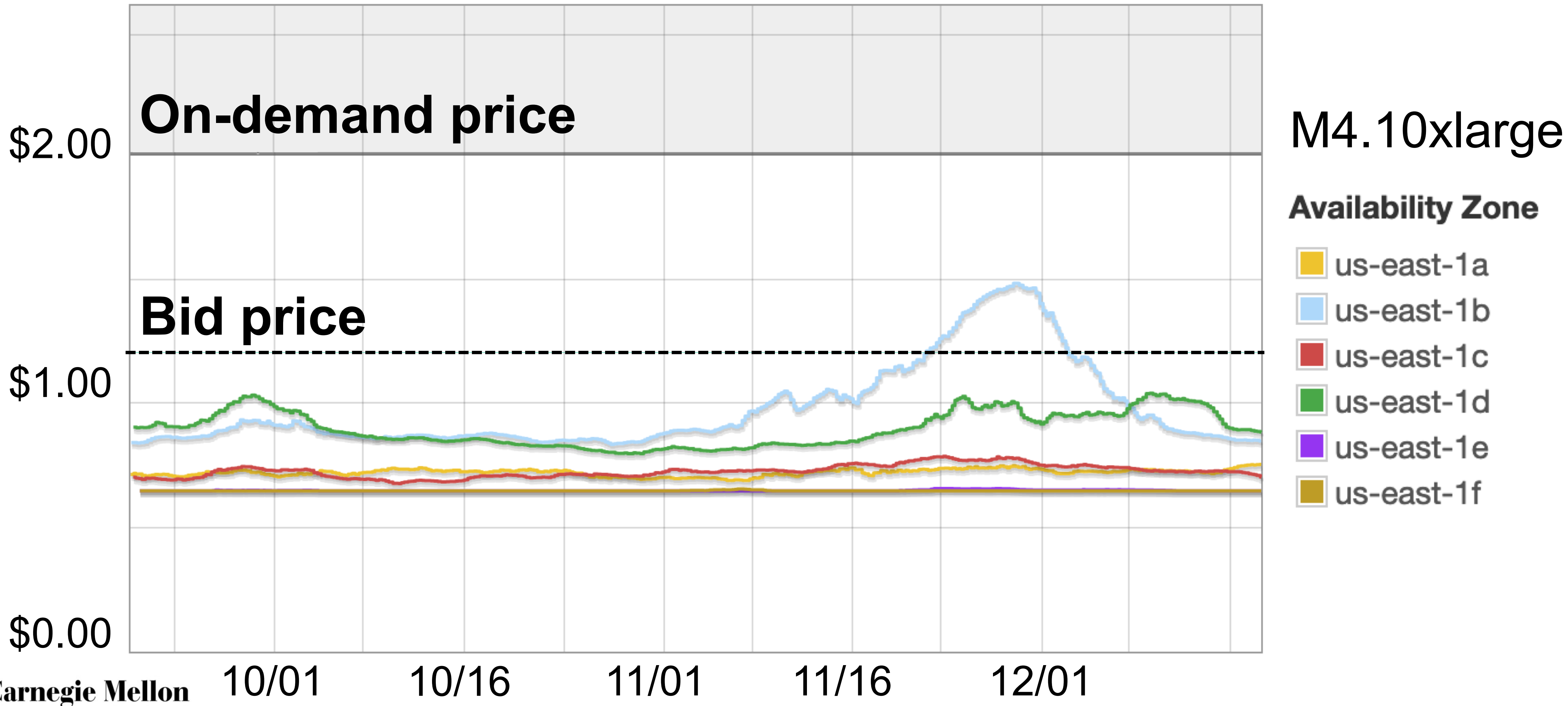
Adv: Often > 50% cheaper vs on-demand, refund if revoked in 1st hr



M4.10xlarge
in us-east-1b

Transient/spot instances in AWS EC2

Adv: Often > 50% cheaper vs on-demand, refund if revoked in 1st hr



Application-specific resource acquisition: Case studies

1. Elastic web services
   - Spot-dancing for elastic services with latency SLOs
   - Tributary [USENIX ATC 2018]
2. General containerized batch task scheduling
   - Cost-aware container scheduling in the public cloud
   - Stratus [ACM SoCC 2018]
     - Best student paper award

Elastic web services & spot instances

- Elastic web services
  - Manage a pool of VMs to serve client requests
  - Need to meet latency SLOs
    - e.g., complete request within X milliseconds
- Spot instances are *cheaper but riskier* than on-demand:
  - Instances can be revoked, leading to missed SLOs

Tributary embraces risk associated w/ spot instances to achieve lower cost while meeting SLOs

Tributary summary

- Naïve selection of spot instances → bulk revocations
- Tributary handles bulk revocations:
  - Uses different bids within the same spot market
    - Higher/lower bid → less revocation risk/more partial-hours
  - Selects resources from multiple spot markets
    - Markets in same region (diff AZ) may not be correlated
  - ML-based prob model → extra resources acquired
    - Added benefit: soaks up unexpected spikes
    - Cost offset by using lower cost VMs and free partial hours

Tributary experimental results and takeaway

- Experimental results
  - Compared systems
    - AutoScale [AWS], ExoSphere [Sharma ‘17], Proteus [Harlap ‘17]
  - Cost reduction by > 21%, decrease SLO misses by > 31%
  - Reduces cost by > 47% for same SLO attainment
- Takeaway
  - Diversified resource pools to mitigate revocation risk
    - Probability model → diverse + extra resources → SLO attained
  - Expected cost + free partial-hours → lower cost for SLO attained

Application-specific resource acquisition: Case studies

1. Elastic web services
   - Spot-dancing for elastic services with latency SLOs
   - Tributary [USENIX ATC 2018]
2. General containerized batch task scheduling
   - Cost-aware container scheduling in the public cloud
   - Stratus [ACM SoCC 2018]
     - Best student paper award

Background and motivation

- Virtual cluster (VC) scheduling:
  - Schedule containerized batch tasks on to rented VMs
    - IaaS CSPs provide a diverse mix of VM offerings
  - Different from traditional cluster scheduling:
    - Add/remove VMs any time → dynamically sized
    - VC can be highly heterogeneous

Stratus takes advantage of diverse offerings and VC elasticity to lower cost of executing batch workloads

Virtual cluster (VC) scheduling properties

1. Wasted resource-time is wasted money
   - Keys to save money:
     - VMs should be highly utilized while rented
     - Use cost-efficient resources
       - Resource prices may fluctuate e.g., in spot markets
2. Possible to have no task queue time
   - Replaced by VM spin-up time
   - Allows bounded workload latency

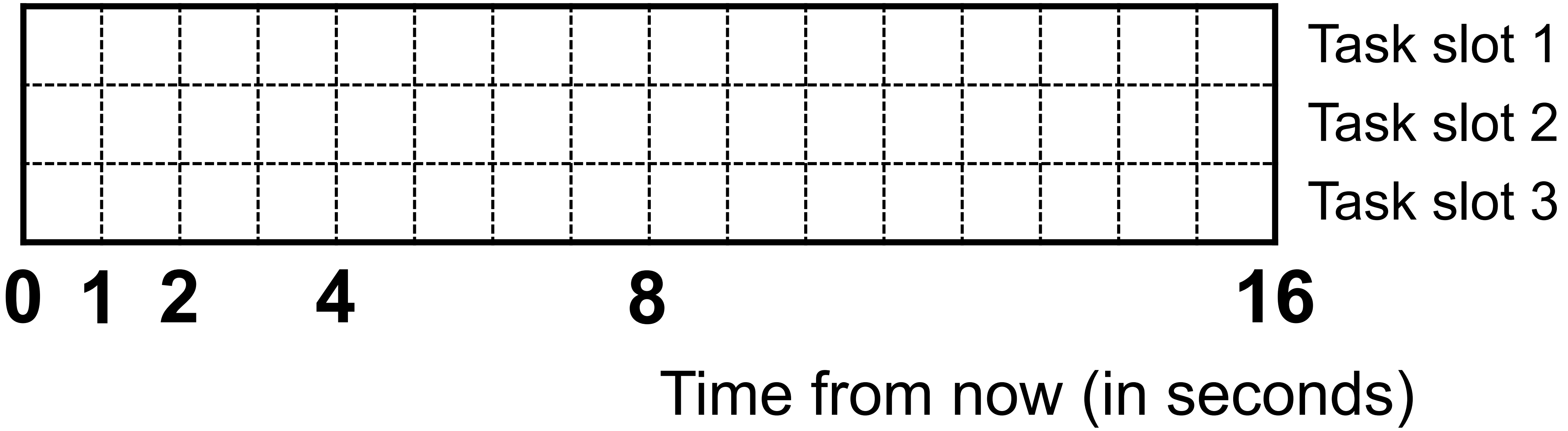
Stratus

- VC scheduling middleware for public clouds
  - Suited for collections of batch jobs
  - How to size VC and where to place tasks
- Goal: Lower cost of executing batch workloads
  - Cost-efficiency by reducing idle VM resource-time
  - Makespan-min by scheduling tasks as they arrive

Runtime binning: Pack tasks of similar runtime on to VM

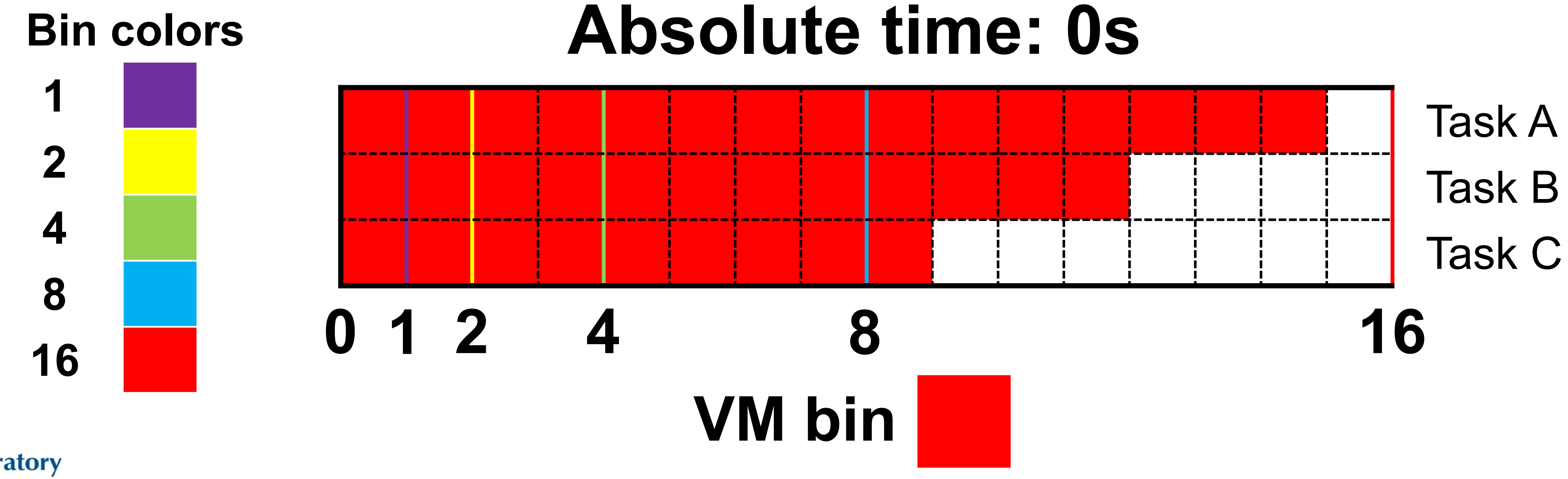
Runtime binning: Notation (simplified)

- VMs specified by “tables”
- Task slots specified by “rows”
- Time-from-now (seconds) specified by “columns”
- Ex: 1 VM, 3 task slots, 16 seconds (0s = now)



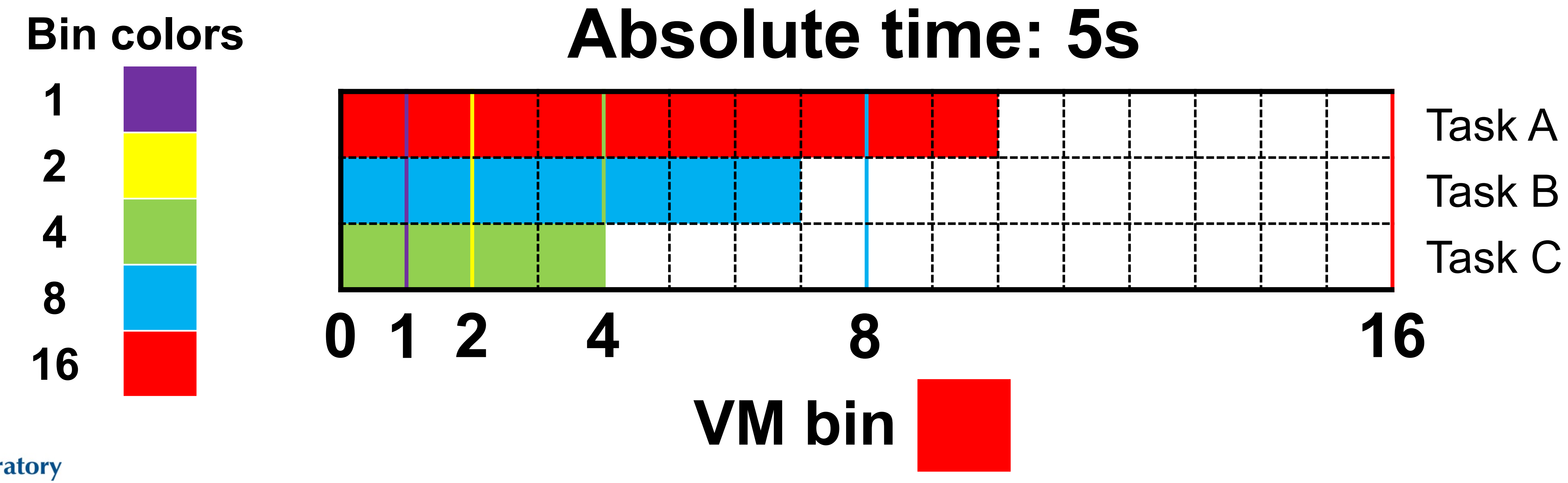
Aligning runtimes: Runtime binning

- Runtime bins: *Logical* groups of tasks and VMs
  - Tasks binned by estimated remaining runtime
  - VMs binned by longest-remaining (estimated) task on VM
- Notation: Runtime bin specified by color



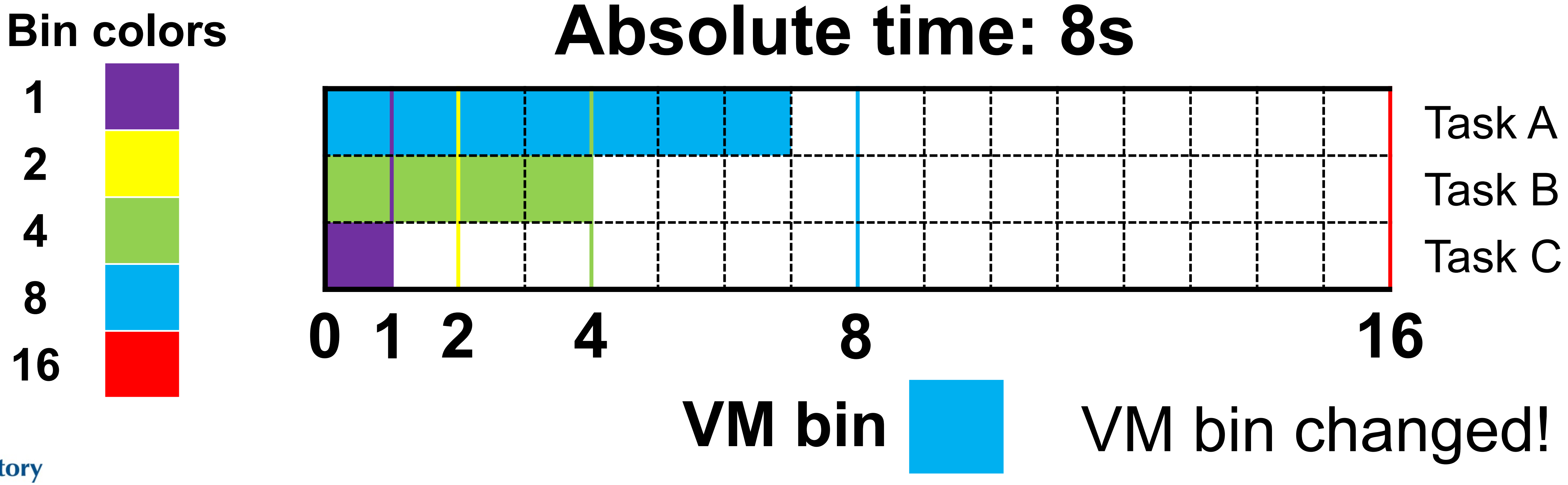
Aligning runtimes: Runtime binning

- Runtime bins: *Logical* groups of tasks and VMs
  - Tasks binned by estimated remaining runtime
  - VMs binned by longest-remaining (estimated) task on VM
- Notation: Runtime bin specified by color



Aligning runtimes: Runtime binning

- Runtime bins: *Logical* groups of tasks and VMs
  - Tasks binned by estimated remaining runtime
  - VMs binned by longest-remaining (estimated) task on VM
- Notation: Runtime bin specified by color



Carnegie Mellon
Parallel Data Laboratory

Packing tasks to VMs

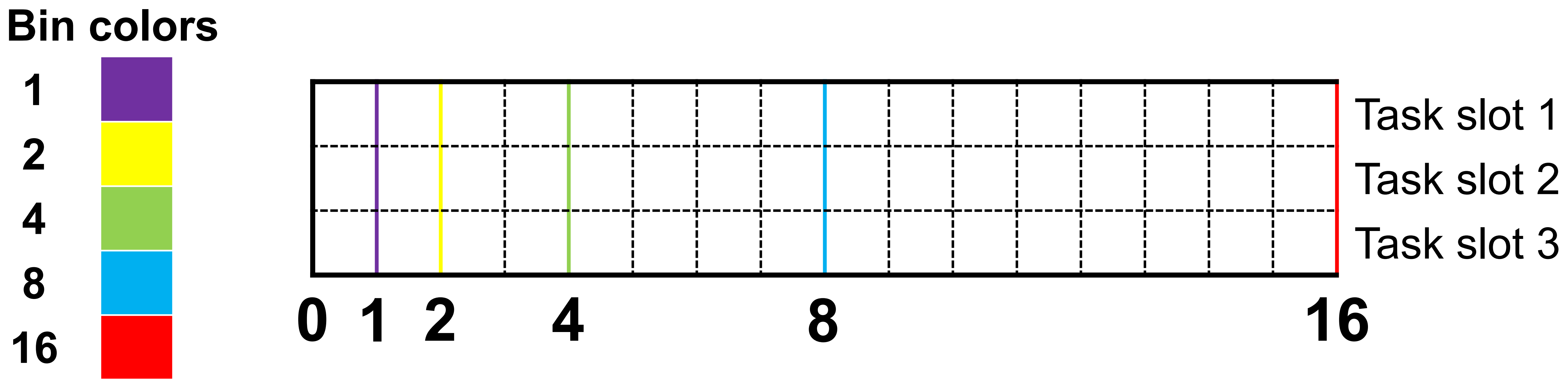
- Each VC manages multiple VMs
  - Each assigned a runtime bin based on its longest-running task
  - VMs released when no tasks running on it
- Packing preference for new task T:
  - VM in T's runtime bin > VM in greater bins > VM in lesser bins
    - Imposes least impact to extend VM time-to-release
  - Only scale out as last resort
- Scaling out:
  - Hypothetical packings + cost-per-resource considerations

Example: Packing tasks to VMs

- VM in T's runtime bin > VM in greater bins > VM in lesser bins
- Least impact to extend VM time-to-release
- Only scale out as last resort

Exponentially-sized runtime bins

- Longer tasks:
  - Greater mis-estimates in absolute
  - Greater straggler effect in absolute
- Short tasks fill in “gaps” as long tasks complete out-of-sync

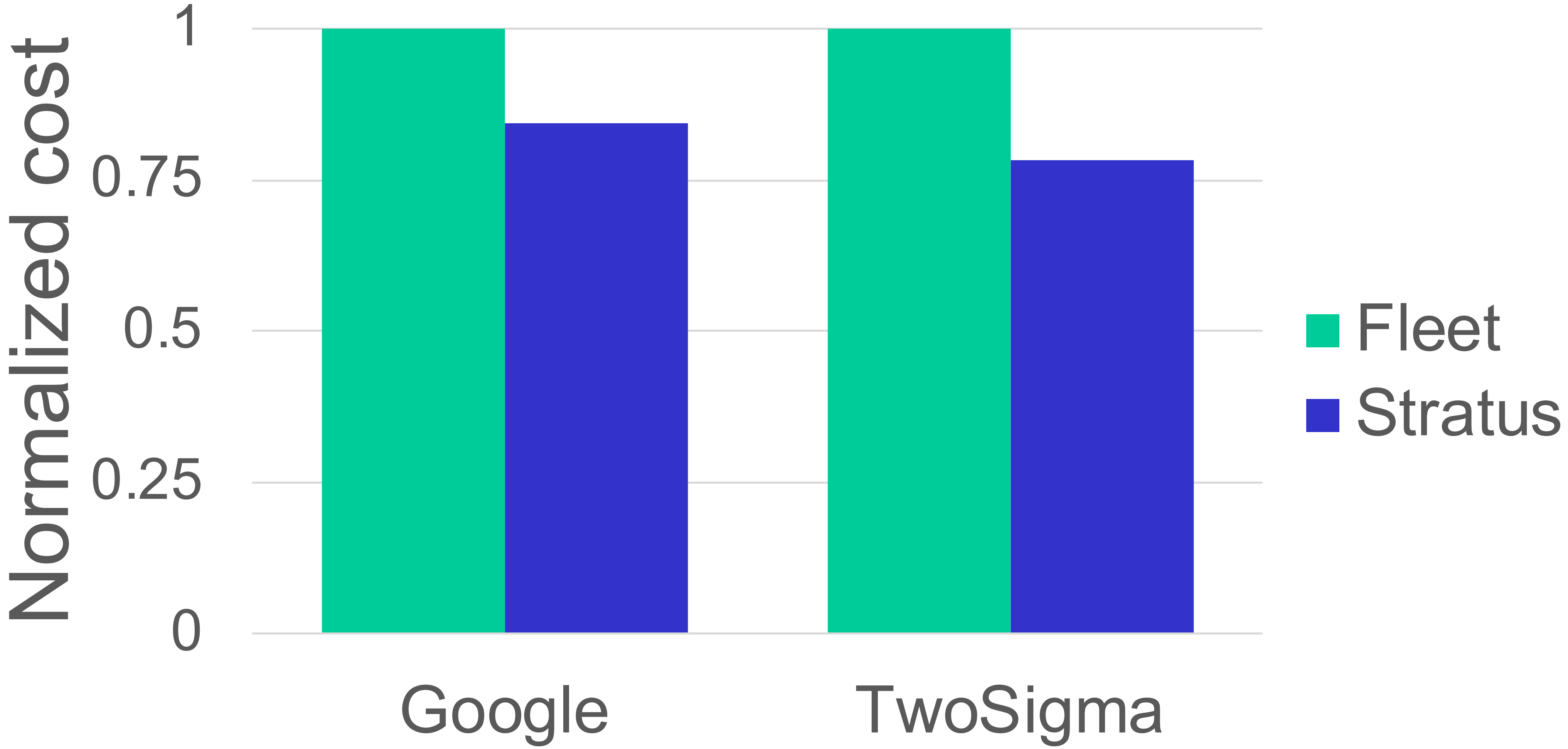


Experimental setup

- Simulation-based experiments
  - Google and Two-Sigma cluster traces
- Task estimates with JVuPredict
  - Modified, aggregate stats-based job runtime predictor
- Focus on batch jobs
  - Filter out jobs running > 1 day
- Spot market traces for dynamically priced VMs
  - Always bid on-demand price – little to no preemptions

Stratus vs Fleet

- Fleet: SpotFleet + ECS (Amazon offerings)
- Stratus reduces cost by 17% (Google) and 22% (TwoSigma)



Stratus takeaway

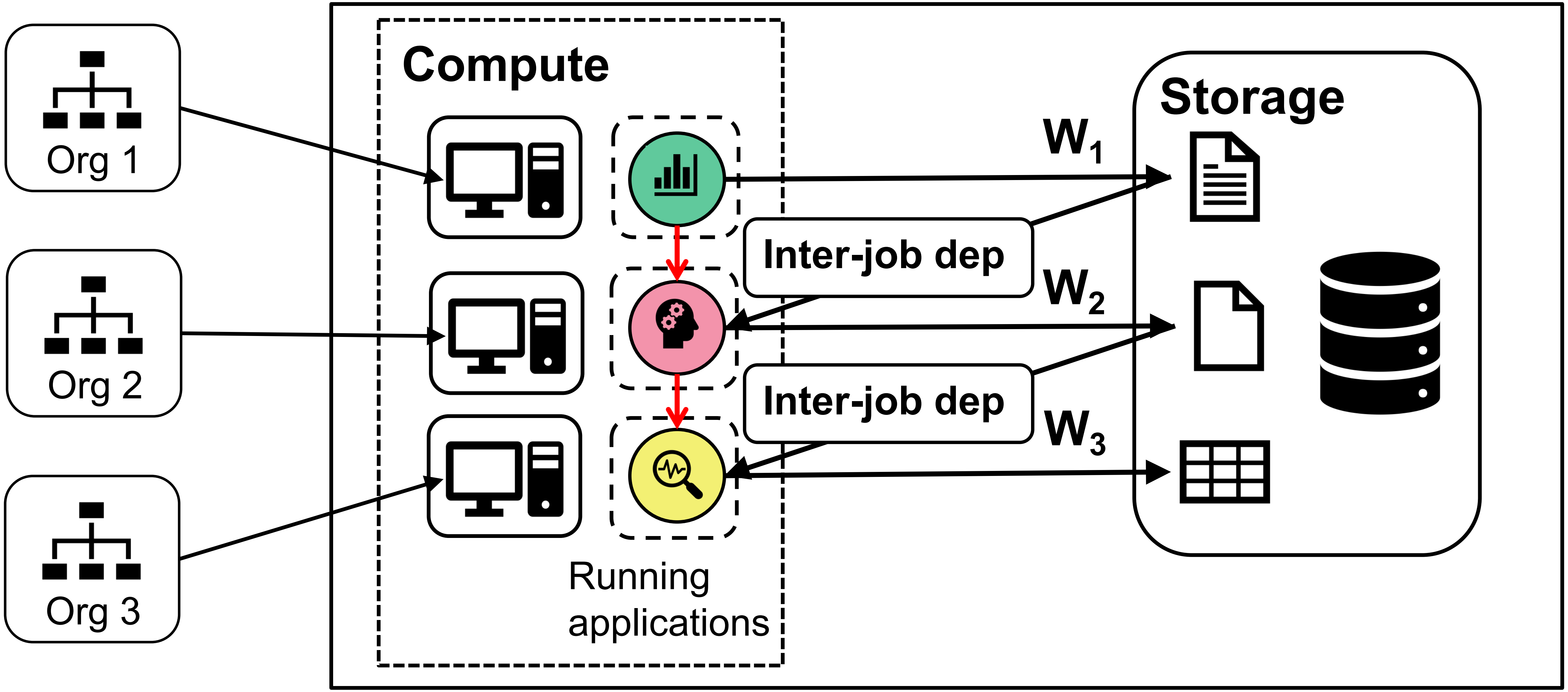
- Runtime binning → high VM utilization during rental
- Simultaneous consideration of scaling, packing, and cost-per-resource leads to reduced cost
- Reduces cost by at least 17% vs other solutions
  - SpotFleet [AWS], HotSpot [Shastri '17], SuperCloud [Jia '16]

Realizing value through dependency-aware resource management

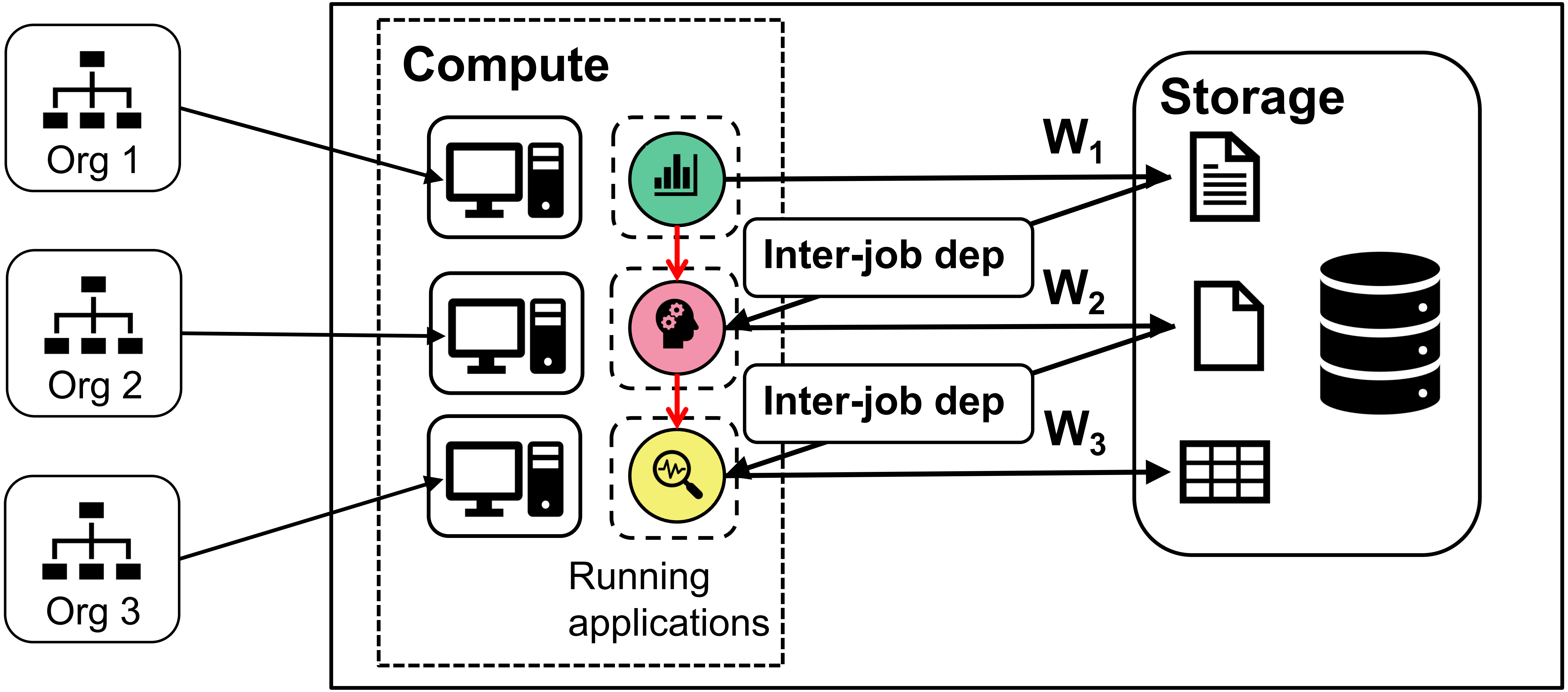
Background: Cosmos

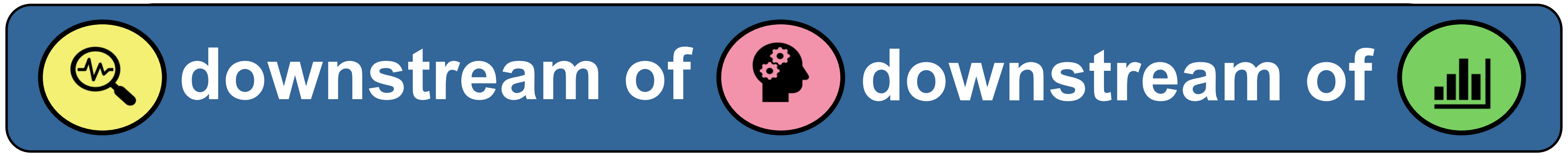
- Microsoft's internal big data analytics platform
- Data from single Cosmos cluster (> 50k servers)
- Mostly batch analytics jobs
    - e.g., Spark, MR-like jobs
    - > 80% dedicated capacity
    - Over 3 months: > 4 mil batch jobs submitted
- Many recurring jobs (> 65%)
    - Jobs that are submitted many times over time

Sharing data → job dependencies



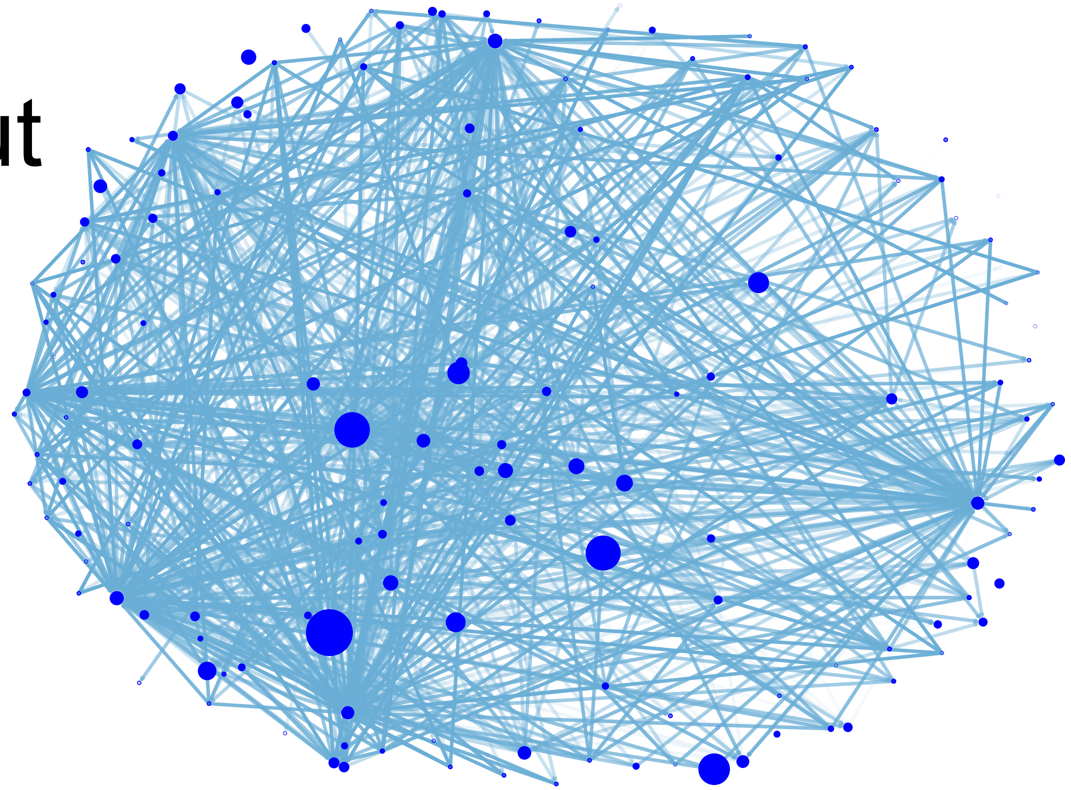
Sharing data → job dependencies





Inter-job dependencies are prevalent!

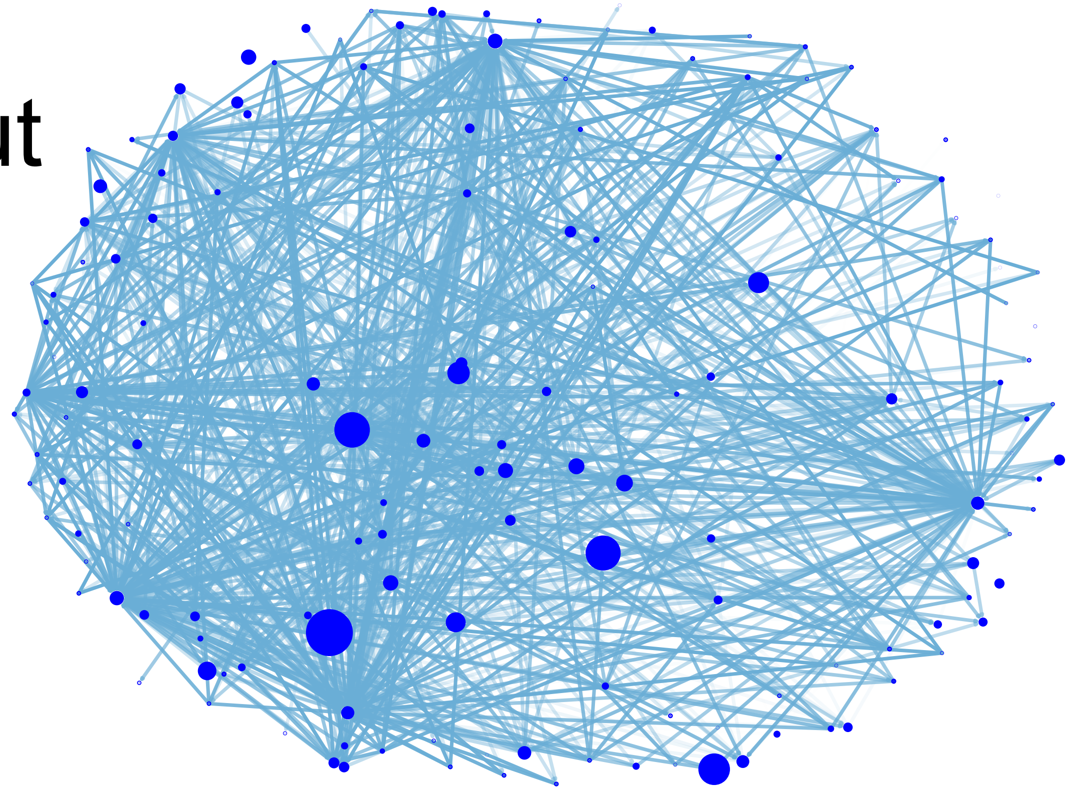
- Of the 4 mil batch jobs analyzed:
  - > 80% of jobs dep on another's output
  - ~16 mil dependencies
    - ~79% deps are recurring
  - 95% of orgs rely on data generated by jobs of another org
    - Often no coordination



Dependencies are very prevalent;
but, we know very little about what they look like

Inter-job dependencies are prevalent!

- Of the 4 mil batch jobs analyzed:
  - > 80% of jobs dep on another's output
  - ~16 mil dependencies
    - ~79% deps are recurring
  - 95% of orgs rely on data generated by jobs of another org
    - Often no coordination



Inter-job dependencies present a whole new avenue that system designers can explore!

Ongoing and proposed work

- Analysis of inter-job dependencies
  - Visualization [SIGMOD 2019 demo]
  - Observations + characterization [Under submission]
- Proposed: Opportunities using inter-job dependencies in resource management to realize more user value
  - Better job valuation → better job scheduling

Job valuation and scheduling

- Manual job priority assignments in most prod clusters
  - Used to determine job resource acquisition order
  - Want: Assignments to reflect job's monetary value
  - Or at least: Principled, consistent assignments
  - Realistically: Manual priority assignments are *unreliable*!
    - 26% recurring dependencies set with inverted priorities
    - 33% ad-hoc jobs w/ priority > avg recurring job
      - Many recurring jobs are production jobs

Opportunity: Deps can help w/ valuation

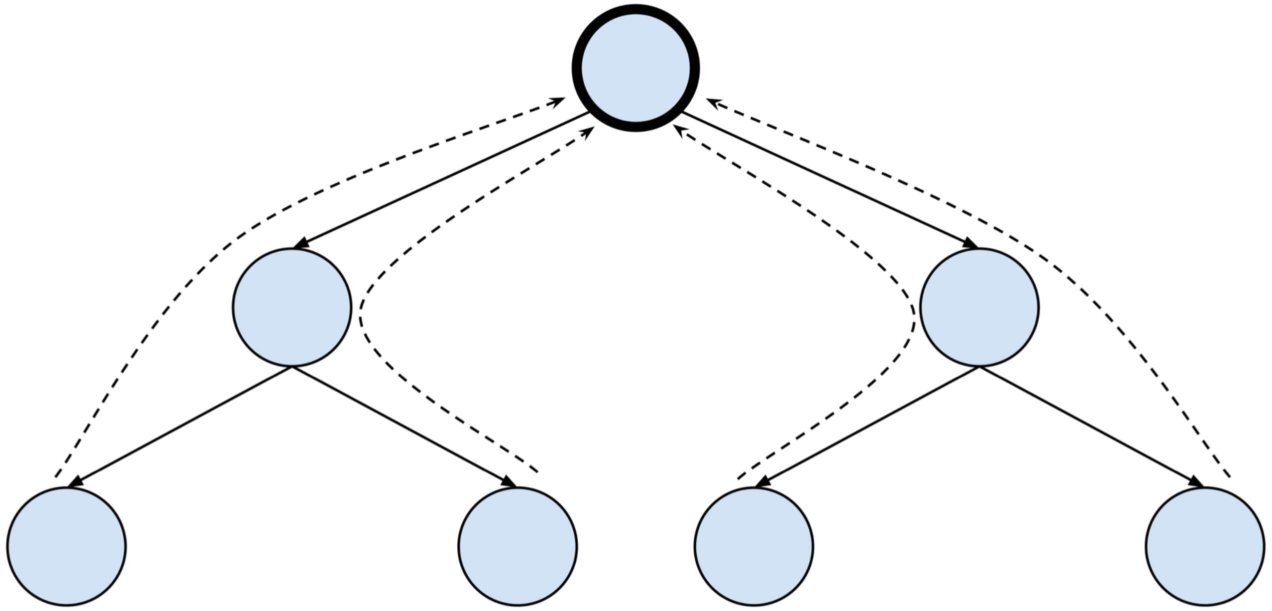
- > 50% of jobs connected in a single dependency subgraph
- 28% recurring jobs are submitted "input-blind"
  - i.e., downstream job requires upstream output to run, but *no coordination* between up/downstream
- Most subgraphs have more leaves than roots:
  - 83x more leaves than roots in largest subgraph

Some jobs more impactful than others:
Historical job dependencies + telemetry → expose job value

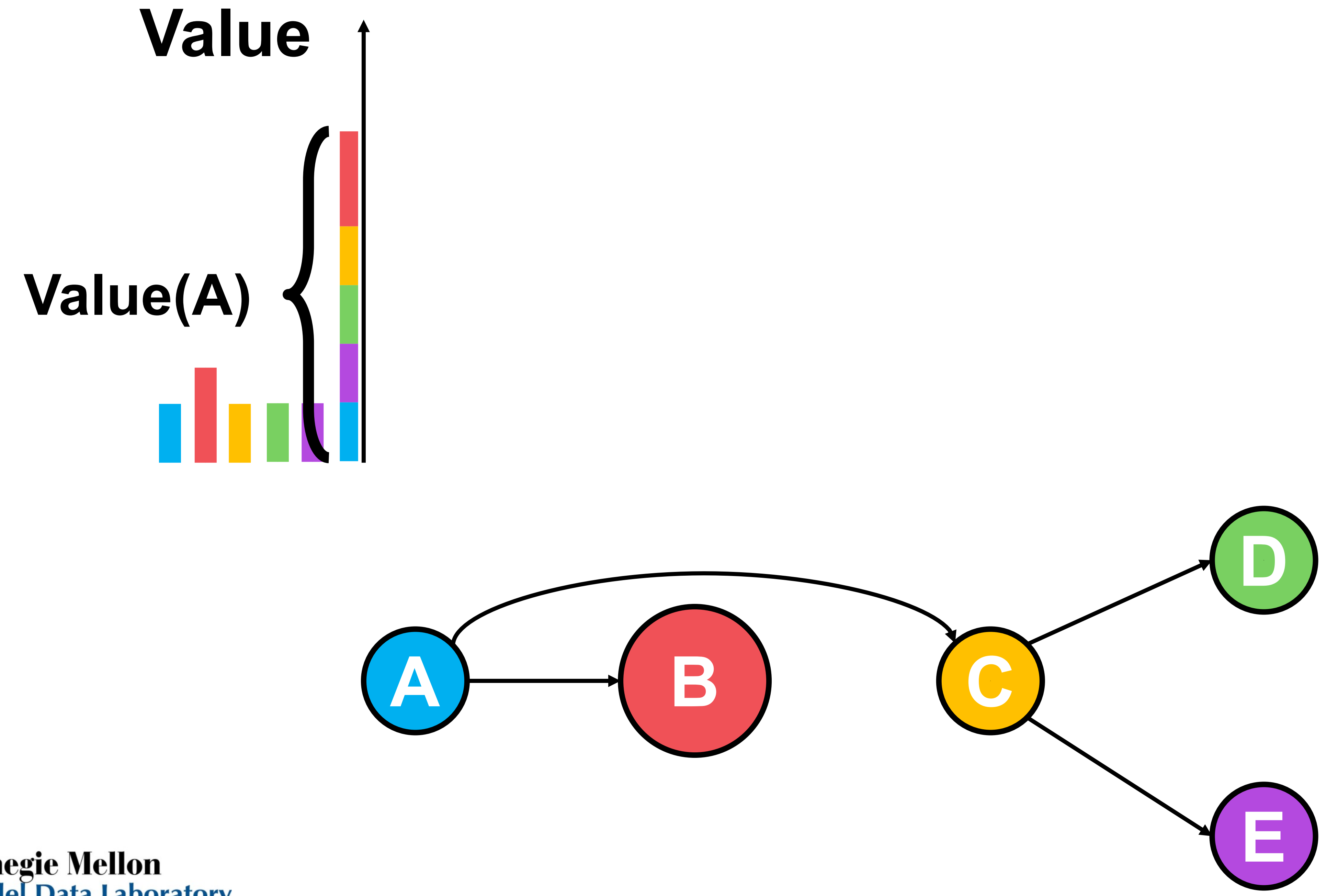
Value flow: Overview

- **Job valuation target:**
  - Recurring jobs (> 65% of jobs)
- **Assumption:**
  - Each job has “inherent value”
    - User- and job-metrics as proxy
- **Value for each job:**
  - Aggregated value of job and its downstream jobs
    - Downstream jobs “contribute” value upstream

If each job 10 downloads,
user-value(root) = 70 downloads



Value flow: An example



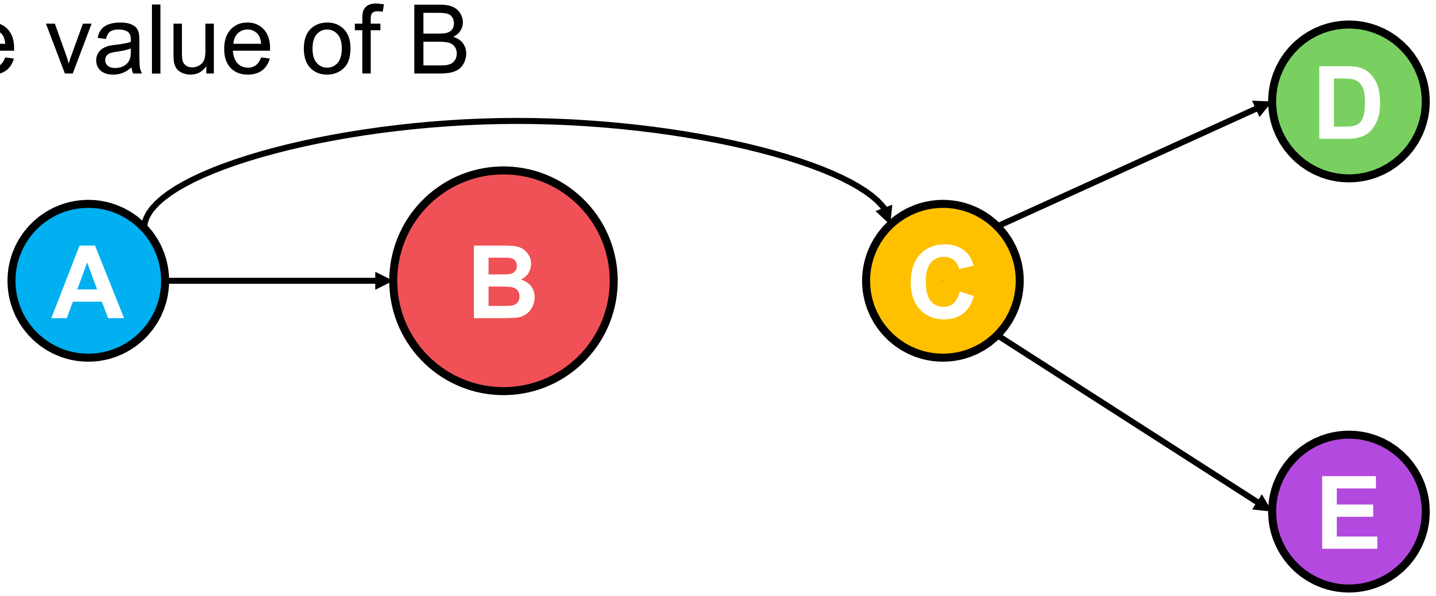
Evaluating dependency-based valuation

- 6 highly important recurring jobs
  - Curated by RM team, (should) have very high assigned priority
- Ranking with Owl's heuristic
  - Metric: Aggregate downloads as value-proxy
  - 5 / 6 in top 4% of jobs, one outlier at top 11%
  - 4 / 6 with relative ranking within 5% of priority-based rankings
    - One ranked 50% higher vs priority-based (49th %ile by priority)
    - One ranked lower than priority-based by 11%
- Mostly consistent w/ priority-based for very important jobs

Utility functions from inter-job deps

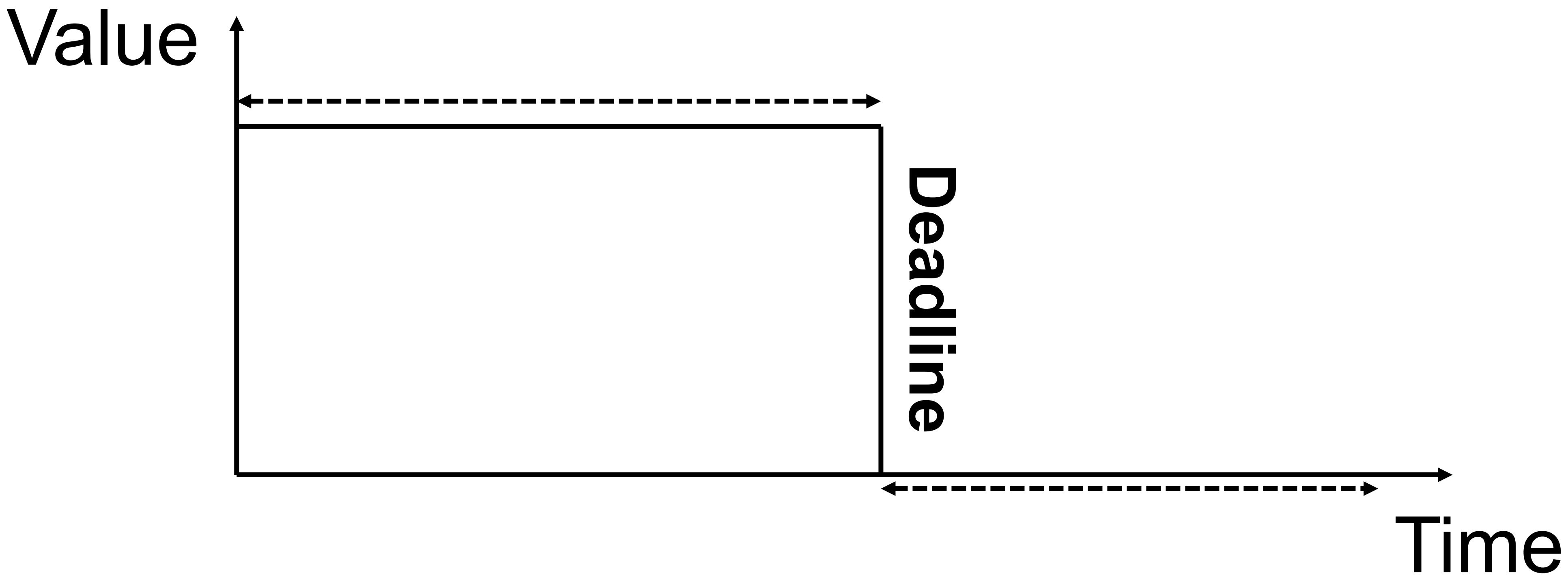
- Job value → priority may not be good enough
  - Better: Value "realizable" by completing job at time T
- Example:
  - Let priority(A) = value(A) = sum all value downstream
  - If A late, downstream (e.g., B) can fail w/ missing input error → A may not always realize value of B

Address with dep-driven utility functions



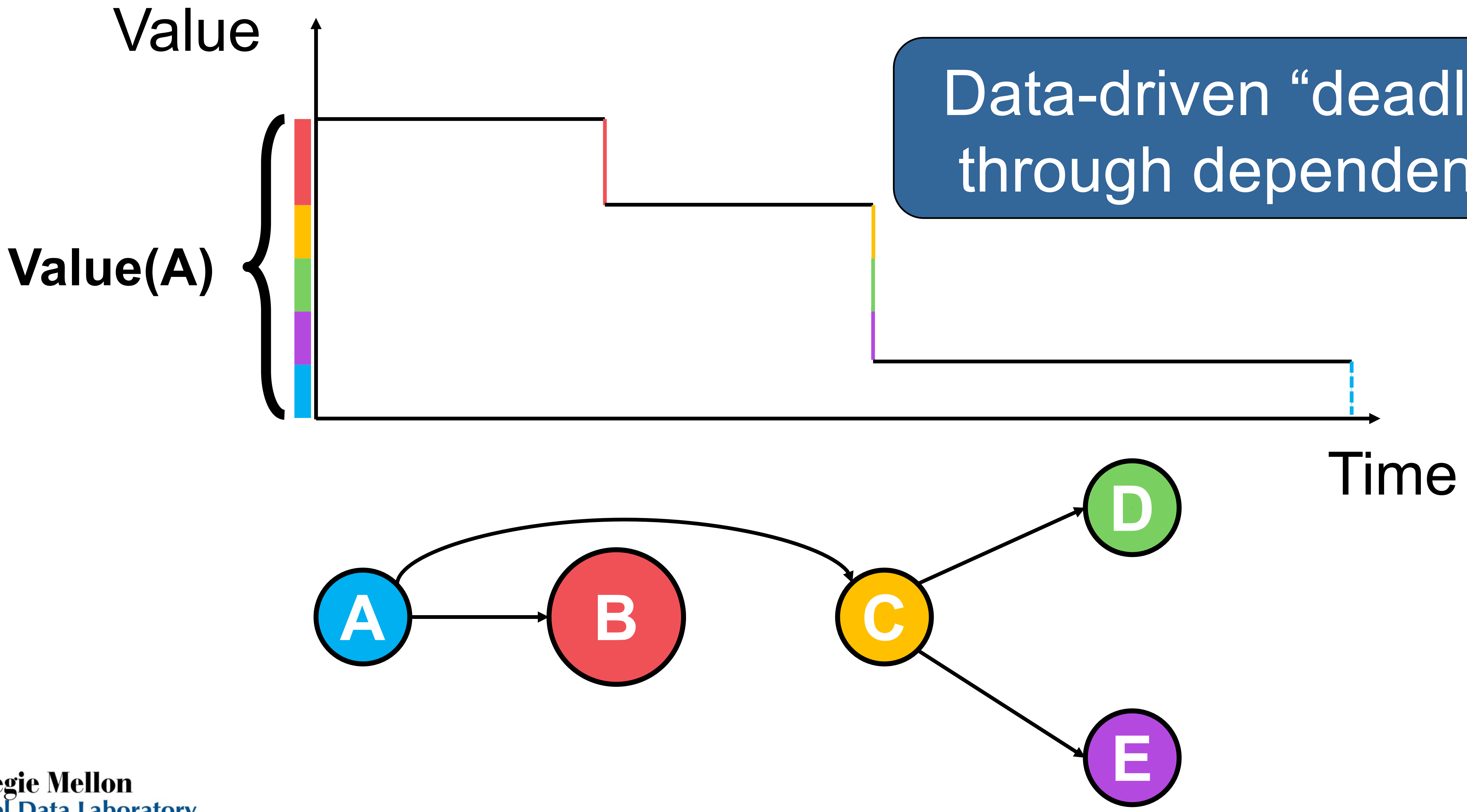
Utility functions

- Utility function
  - Job value as a function of job completion time
- A common representation in scheduling literature:



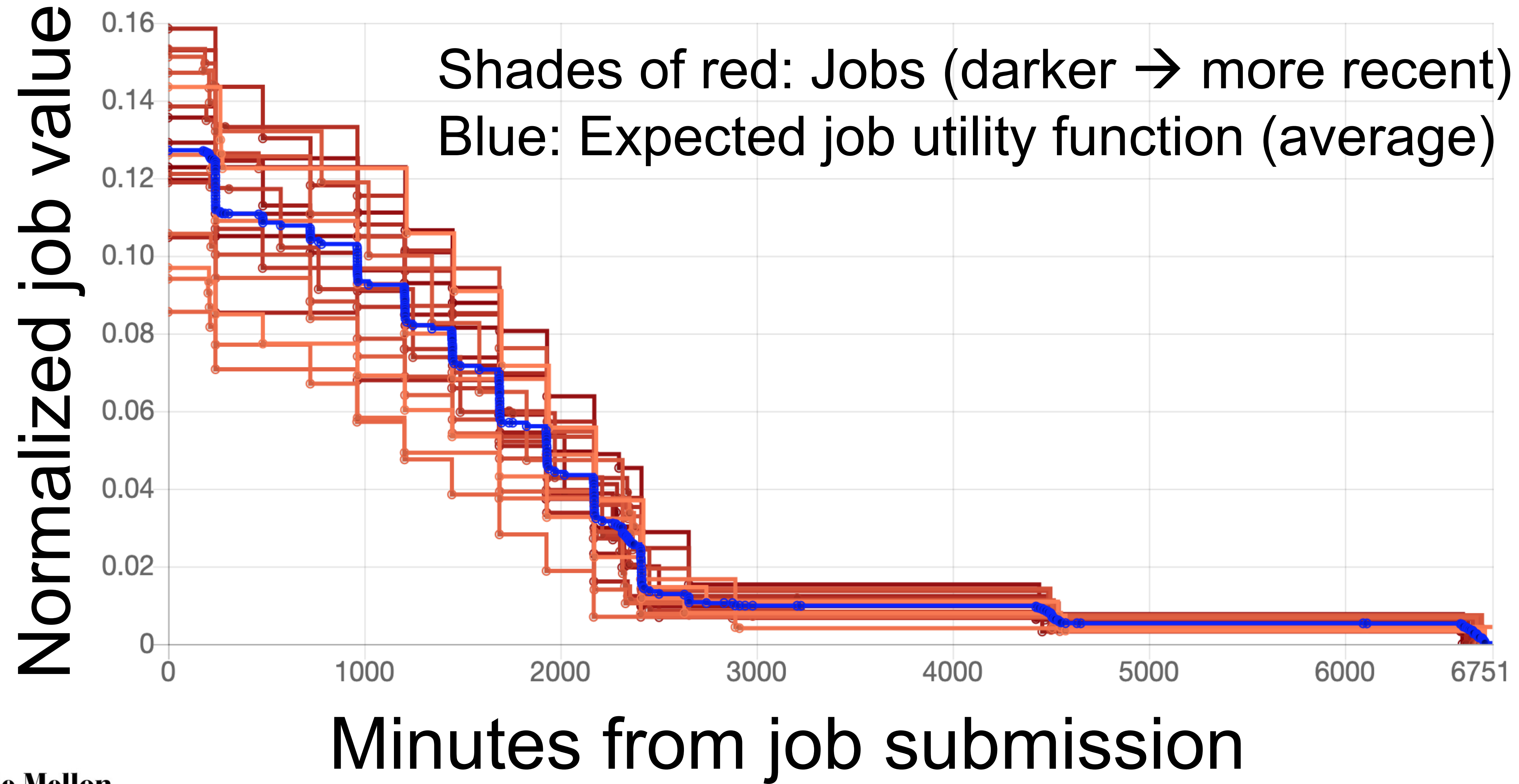
Studied extensively, but none in the wild!

Dependency-driven utility function



Data-driven "deadlines" through dependencies

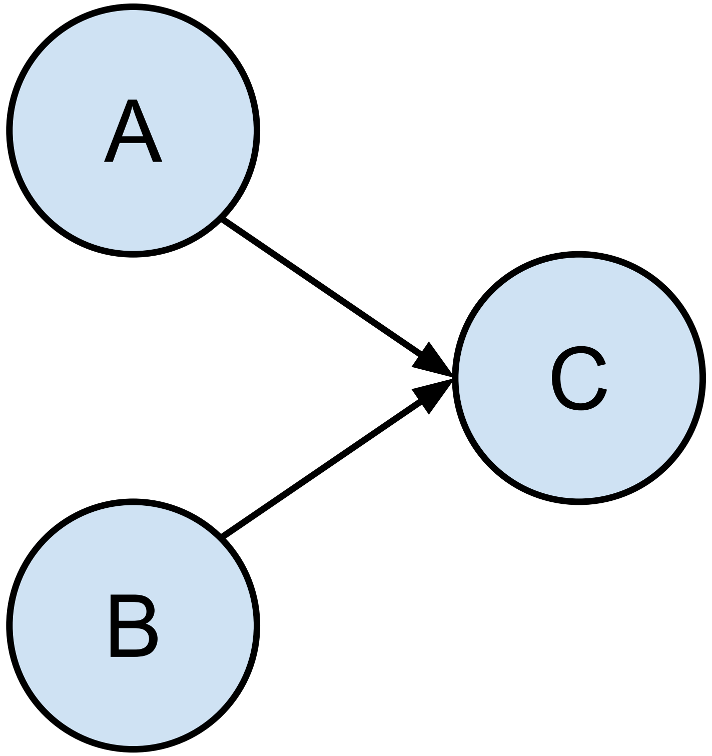
Owl: Real utility function [SIGMOD demo '19]



Carnegie Mellon
Parallel Data Laboratory

Utility function limitations in scheduling

- Utility functions, as a construct, cannot encode:
  - Value when a job has multiple upstream jobs
  - Dependency properties



Opportunity for something better:
Use properties of historical dependencies

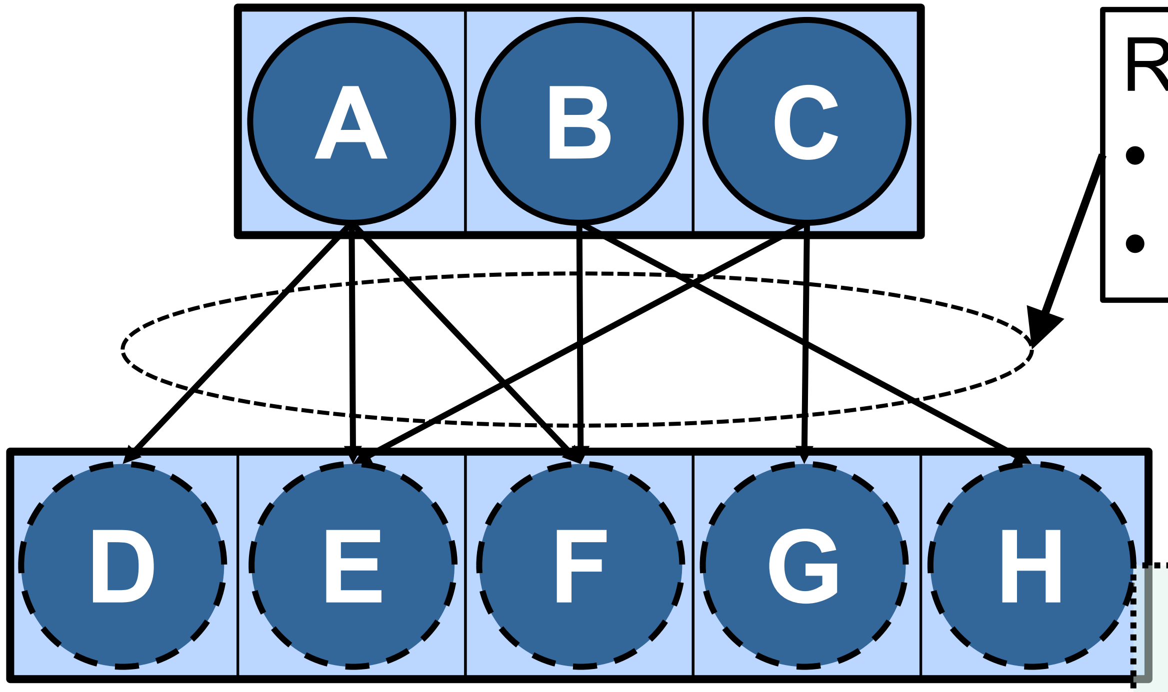
Ongoing work: New data representation?

- **Maybe:** Utility functions not enough in some scenarios
- **Ideal:** For job, want entire historical graph of downstream
    - – Can be expensive if many jobs downstream
- **Idea: Work-in-progress**
    - Keep track of *potential* jobs only one-hop downstream
        - – Probabilistic view of upcoming jobs with historical deps
            - 79% of dependencies are recurring
    - Aggregate info for jobs 2+ hops downstream

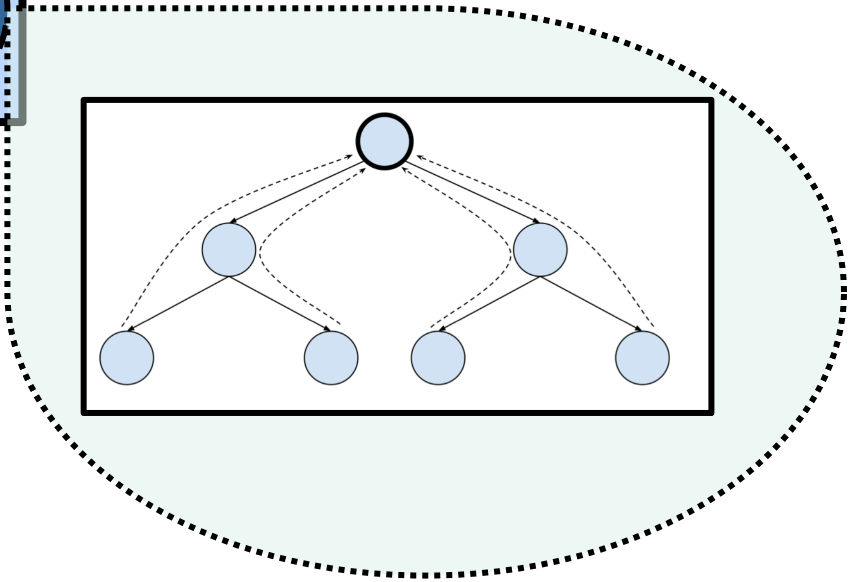
Talon: Scheduling with new data representation!

Job queue

Expected follow-on jobs



Recurring dep metadata:
- Dependency properties
- P(arriving | completed U queue)



Talon: In process of building

Ongoing work takeaway

- Inter-job dependencies → new opportunities!
- Dependency-aware job scheduling
  - Better, data-driven, and hands-free job valuation
  - Dependency-based utility functions
  - New dependency-aware scheduling data representation
- Questions to explore:
  - Is priority (based on historical value) good enough?
  - How expressive are utility functions?
  - Are new data representations better / necessary?

Overview: Contributions & ongoing work

- Realizing value through user application frameworks
  1. Elastic web services: Tributary [USENIX ATC 2018]
  2. General task scheduling: Stratus [ACM SoCC 2018]
     - Best student paper award
- Realizing value through dependency-aware resource management
  - Dependency visualization: Owl [SIGMOD demo 2019]
  - Dependency analysis: Under submission
  - Dependency-aware scheduling: Ongoing work

Proposed thesis timeline

Time	Plan
Nov. 2019	Submission of inter-job dependency analysis paper to EuroSys 2020
Dec. 2019 – Feb. 2020	Design and initial implementation of Talon
	Thesis proposal preparation
Feb. 2020	Thesis proposal
Feb. – Apr. 2020	Experiment design and refinement of Talon
May – Dec. 2020	Finish experiments on Talon and paper submissions (targeting OSDI 2020)
Jan. – May 2021	Dissertation writing, defense, and job search

Thank you!