Distributed Systems

Spring Semester 2020

Lecture 20: Cluster Management (Borg)

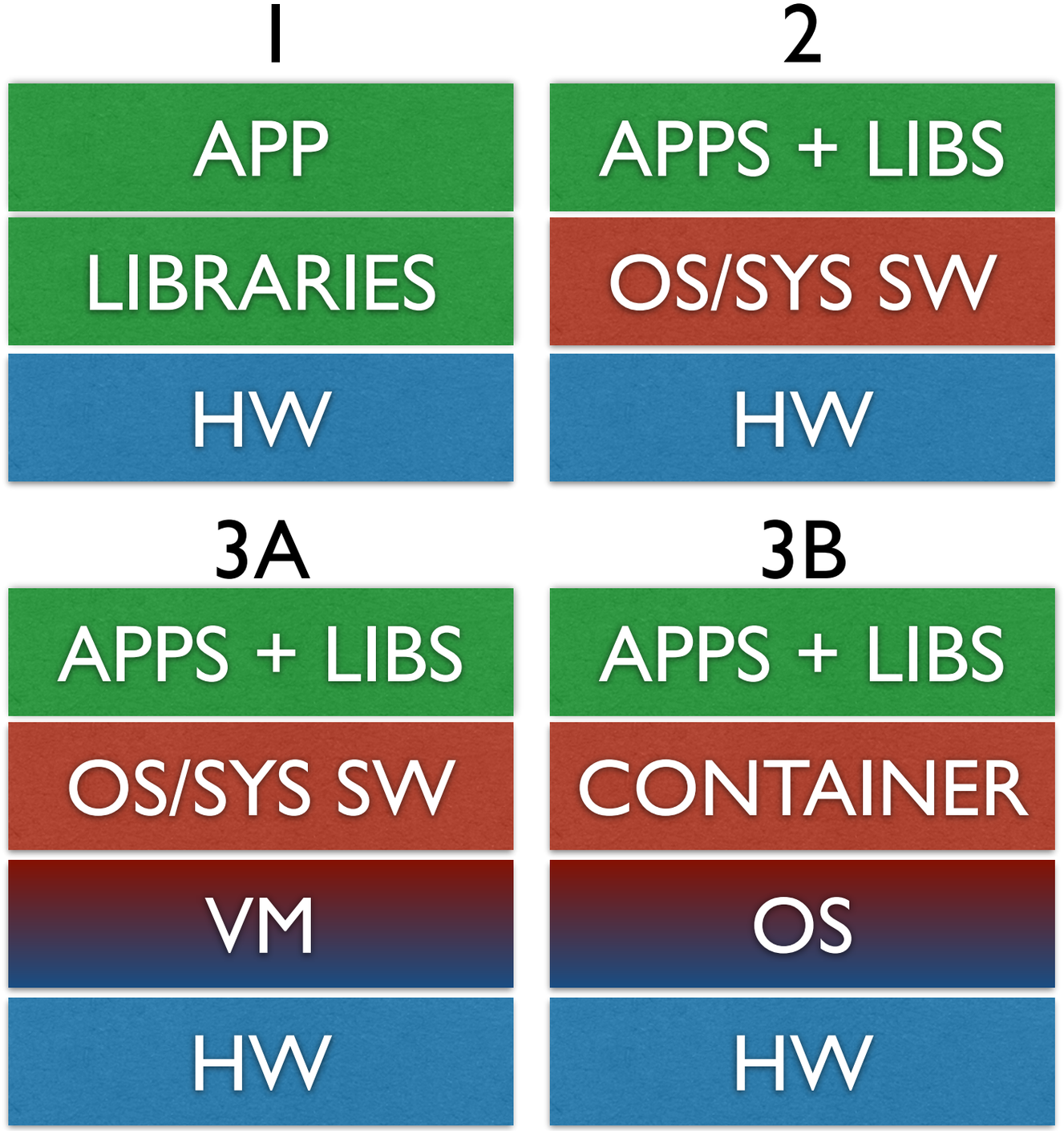
John Liagouris
liagos@bu.edu

Google Borg

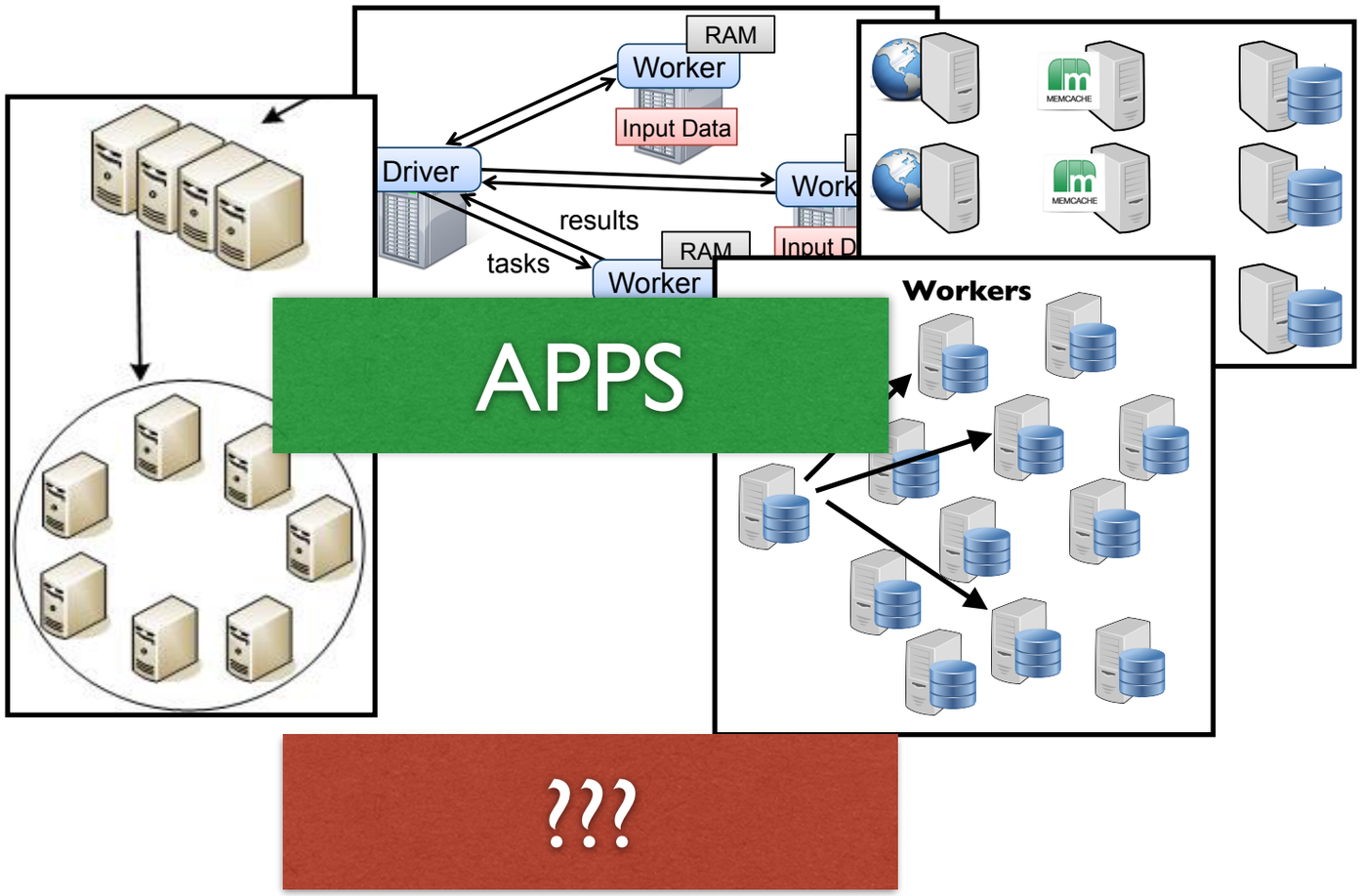
1. Developed at Google for internal use on large, shared clusters
    1. Automates application deployment and resource management on large clusters
    2. Underpins almost all Google workloads, including "cloud" business, MapReduce
2. Inspired several similar open-source systems
    1. Apache Mesos
    2. Google Kubernetes
    3. Docker Swarm
    4. Hashicorp Nomad
3. Hot topic in industry right now ("orchestration")

THE STACK

- Our standard view of computing:
  - Isolate — every program has its own machine but share single machine to improve utilization
- Insulate applications from differences in hardware and OS's
- Package all parts of an app and its dependencies into a sealed image
- Automate deployment



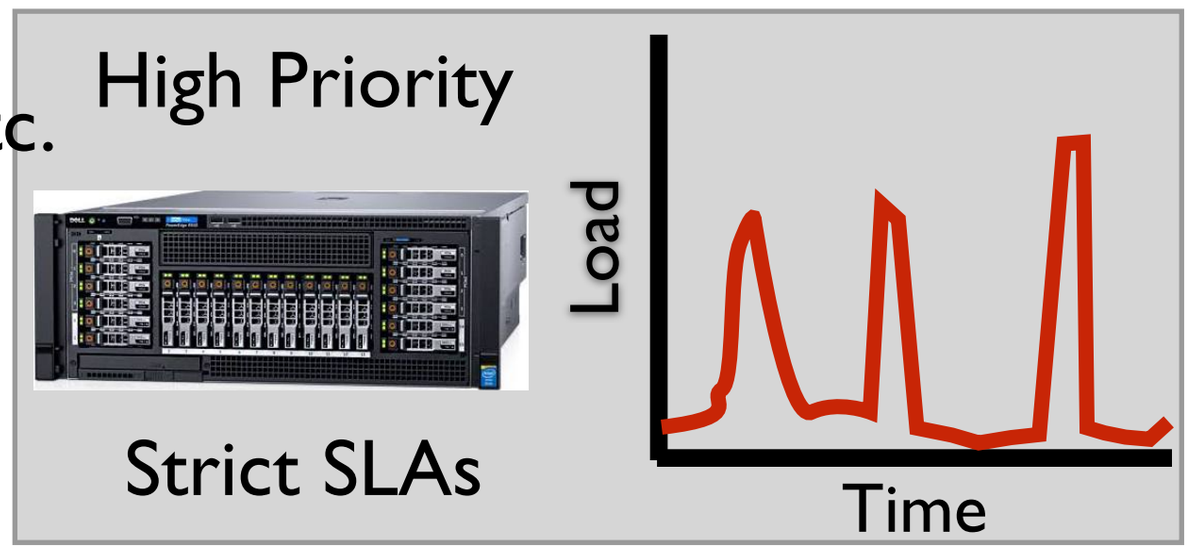
DATA CENTER COMPUTER



- Applications: MapReduce instances, Spark instances, GFS, Webservices, etc.
  - Heterogeneous collection of user binaries distributed over 100's of OS instances
- Hardware: Massive integrated data centers

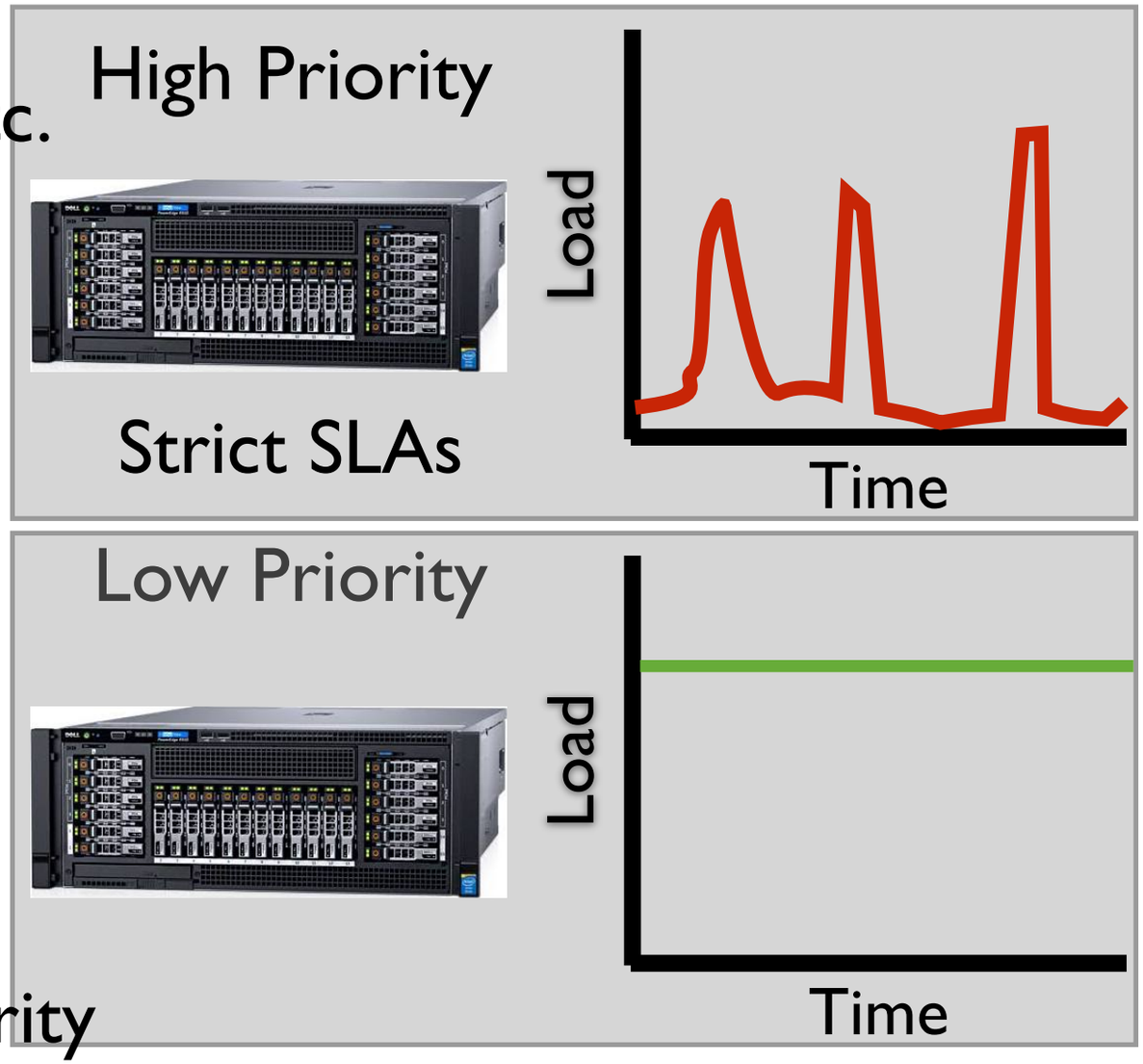
Consolidation = $

- Want to server millions of concurrent users -- Google Doc's, gmail, search, etc. (high priority service -- time sensitive)
  - Must provision for peak load
    - But most of the time
      - load << peak
        - Hence machines are poorly utilized
- But also typically have lots of low-priority batch (time **in**sensitive) work too
- If we can combine the two on a single set of machines by back-fill utilization troughs -- we can get a way with fewer machines in total e.g. 2 goes to 1



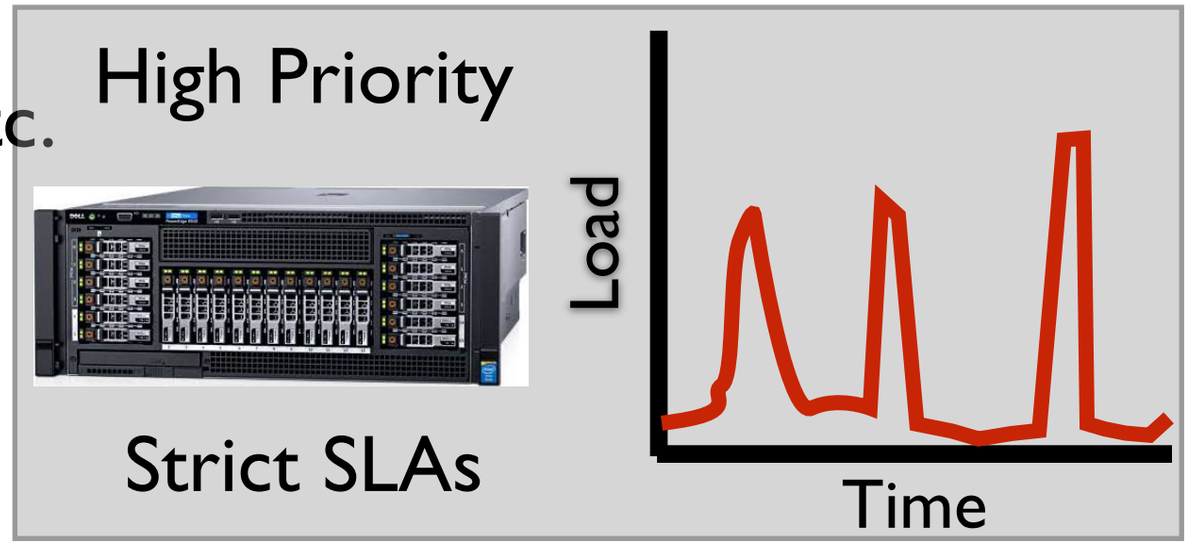
Consolidation = $

- Want to server millions of concurrent users -- Google Doc's, gmail, search, etc. (high priority service -- time sensitive)
  - Must provision for peak load
    - But most of the time
      - load << peak
      - Hence machines are poorly utilized
- But also typically have lots of low-priority batch (time **in**sensitive) work too
- If we can combine the two on a single set of machines by back-fill utilization troughs -- we can get a way with fewer machines in total e.g. 2 goes to 1



Consolidation = $

- Want to server millions of concurrent users -- Google Doc's, gmail, search, etc. (high priority service -- time sensitive)
  - Must provision for peak load
    - But most of the time
      - load << peak
        - Hence machines are poorly utilized
- But also typically have lots of low-priority batch (time **in**sensitive) work too
- If we can combine the two on a single s of machines by back-fill utilization troughs -- we can get a way with fewer machines in total e.g. 2 goes to 1



How about simple time-sharing?

- So can't we just use a cluster of UNIX boxes? -- give each user accounts on all the machines
  - Developer/SysAdmins place apps manually
    - SERVICE: log in, install software, config to start service on init
    - BATCH: log in, install program and start
    - Let OS scheduling take care of the rest!

What could go wrong with this approach?

poor security

- many users have ssh login to machine
- ... but cannot give everyone root (= installing things is painful)
- file system, process list, etc., are shared => information leaks

poor convenience

- shared file system and libraries: version/dependency mess
- user/developer must understand how and where to deploy application

poor efficiency

- manual human configuration required, nothing is automated
- once a service/application is deployed to a machine, it's static

BORG to the Rescue

Borg: an automated cluster manager

Design goals:

Hide details of resource allocation and failure handling

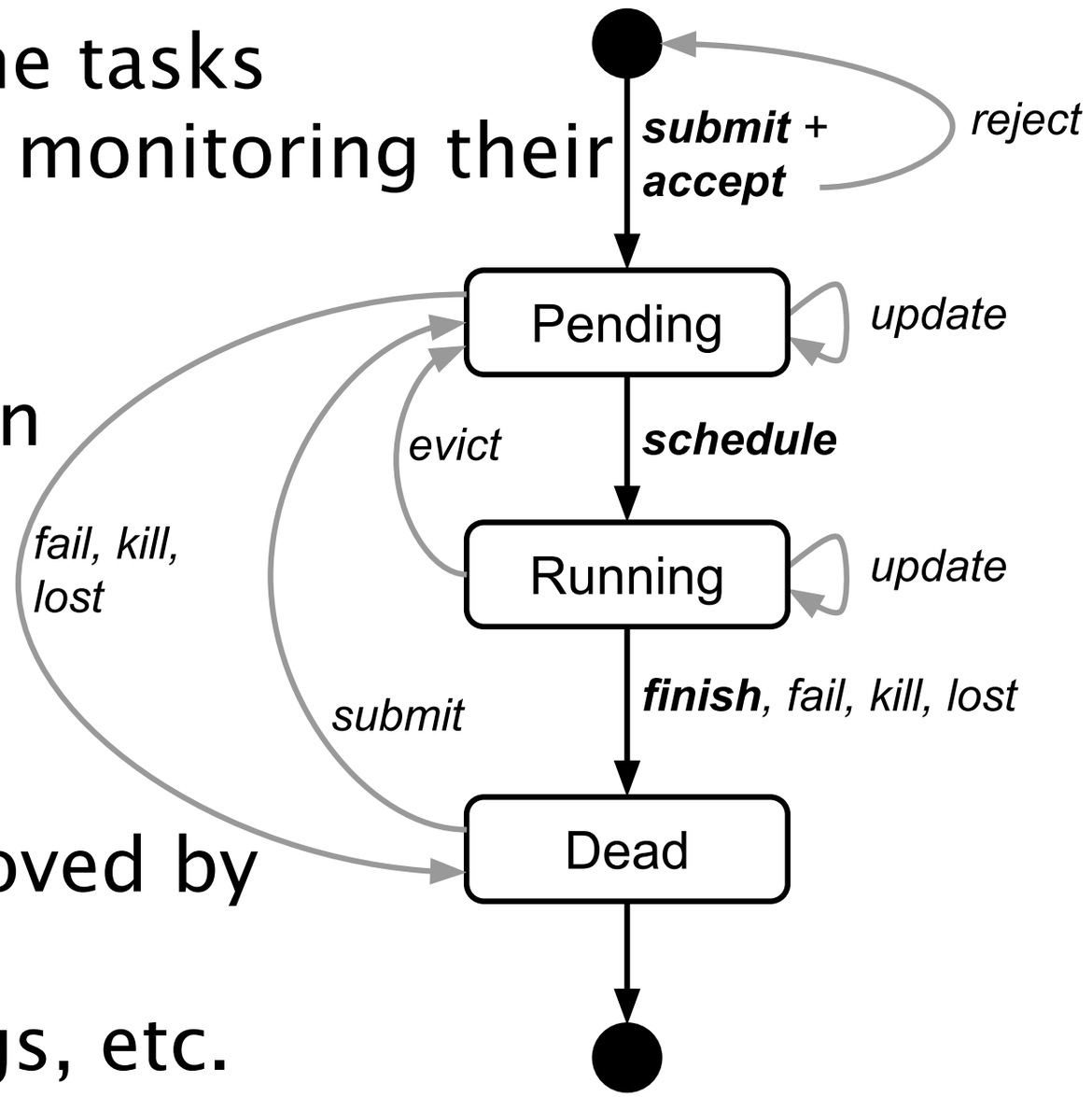
Very high reliability and availability

Efficiently execute workloads over 10,000+ machines

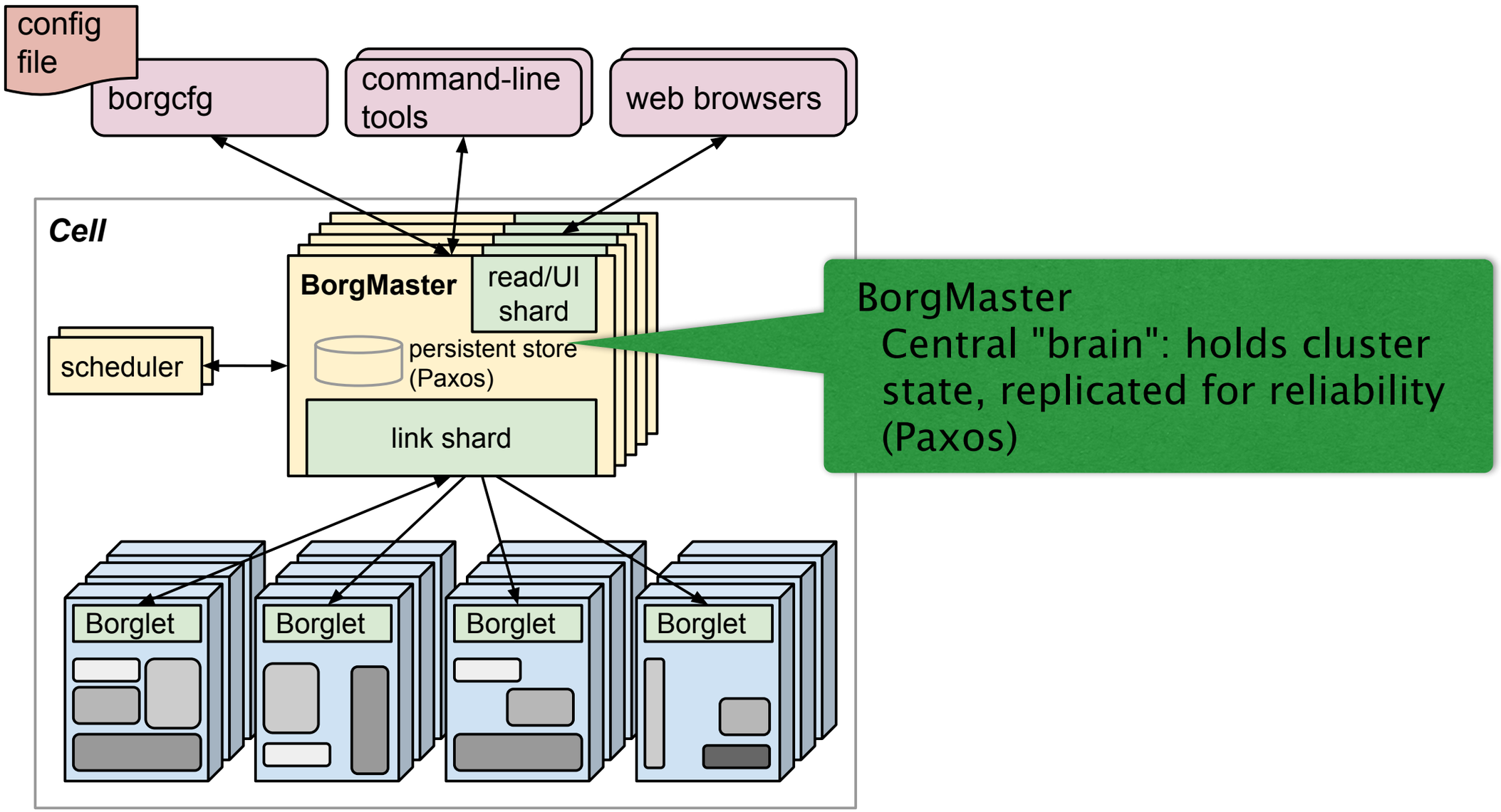
Support *all* workloads
- storage servers (GFS, BigTable, MegaStore, Spanner)
- production front-ends (GMail, Docs, web search)
- batch analytics and processing (maps tiles, crawling, index update)
- machine learning (Brain/TensorFlow)

Borg Lifecycle Overview

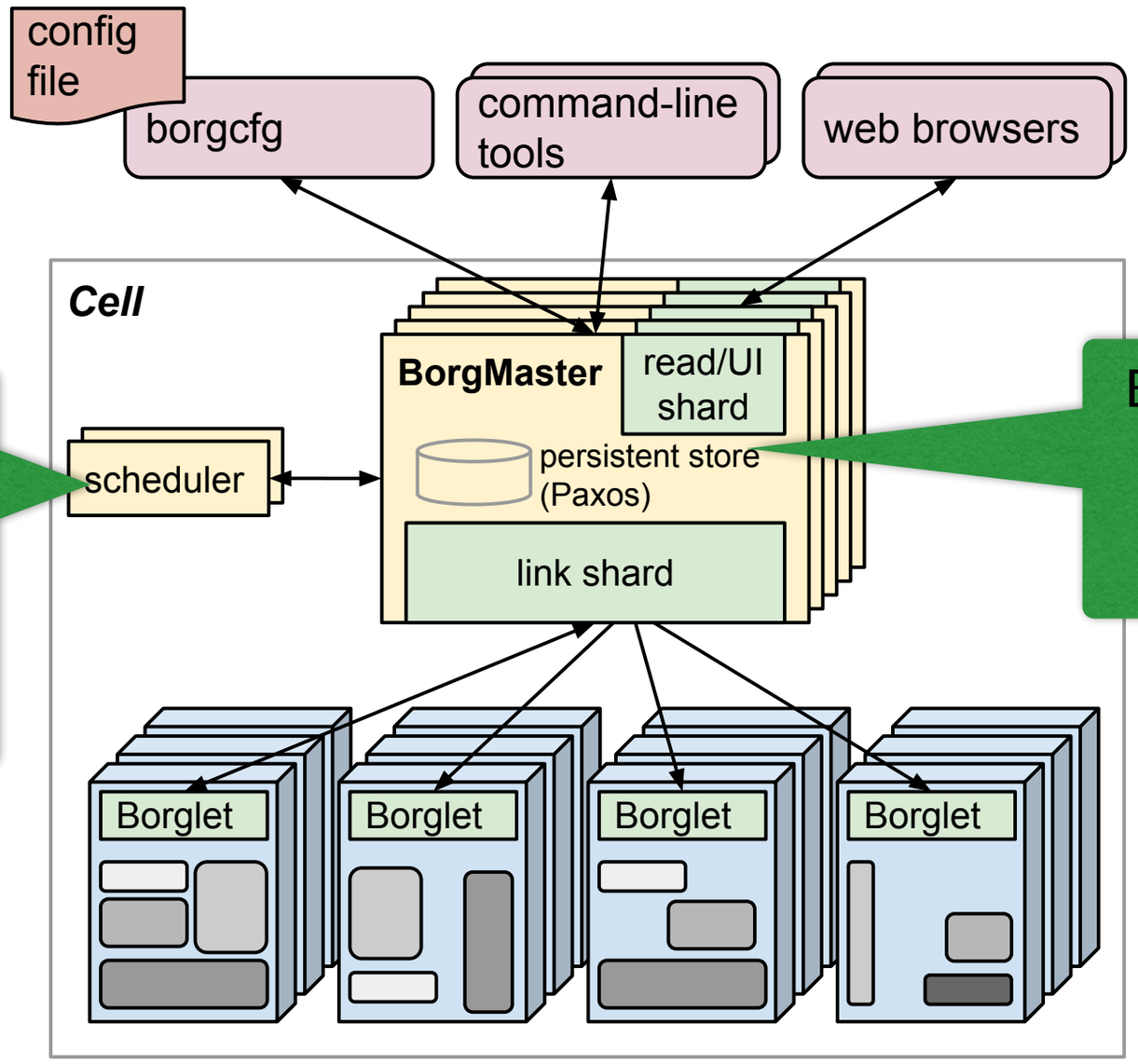
1. User (Google engineer) submits a job, which consists of one or more tasks
2. Borg decides where to place the tasks
3. Tasks start running, with Borg monitoring their health
4. Runtime events are handled
   a) Task may go "pending" again (due to machine failure, preemption)
   b) Borg collects monitoring information
5. Tasks complete, or job is removed by user request
6. Borg cleans up state, saves logs, etc.



Borg architecture



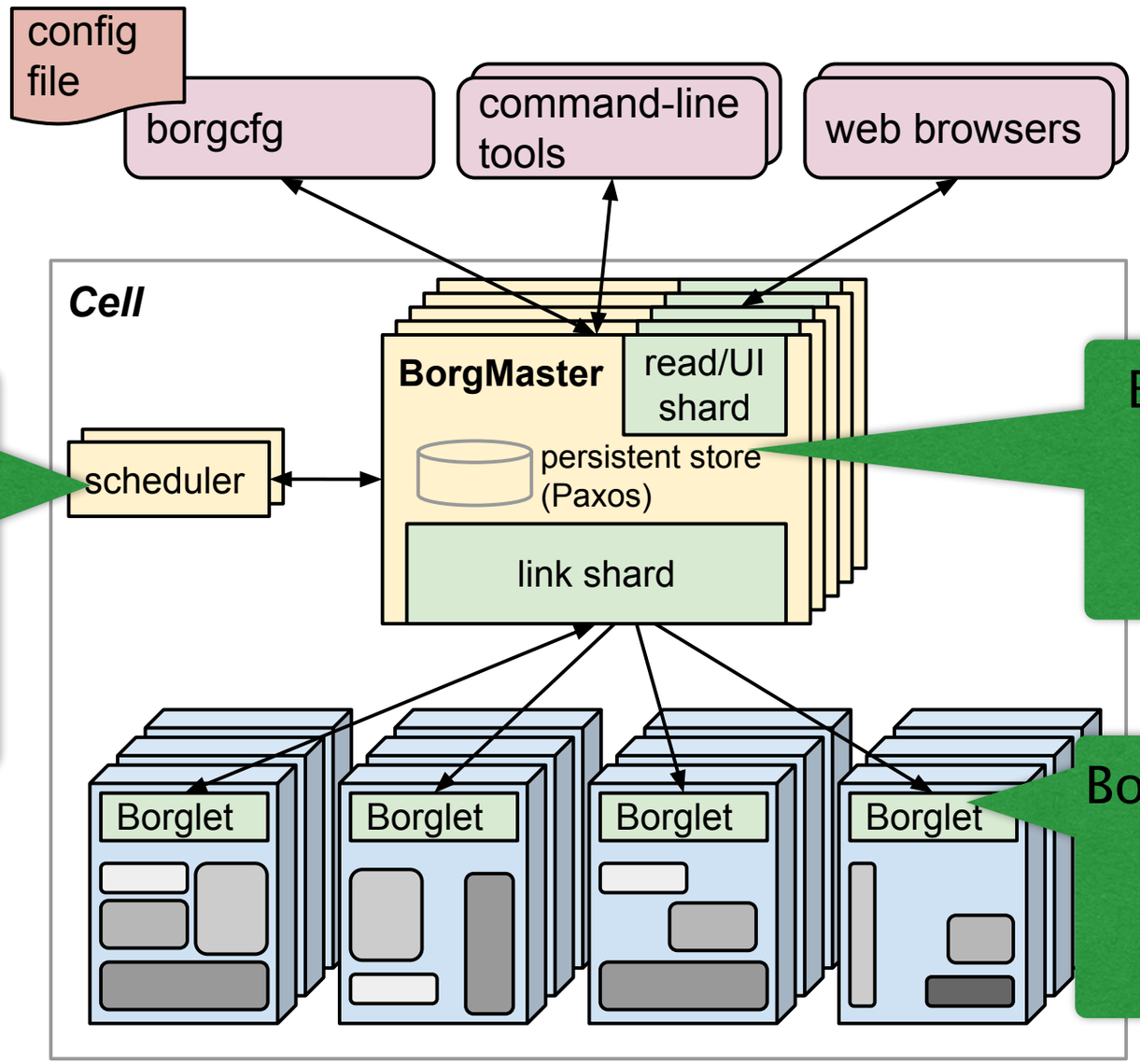
Borg architecture



Scheduler
Decides on which machines task are placed on

BorgMaster
Central "brain": holds cluster state, replicated for reliability (Paxos)

Borg architecture



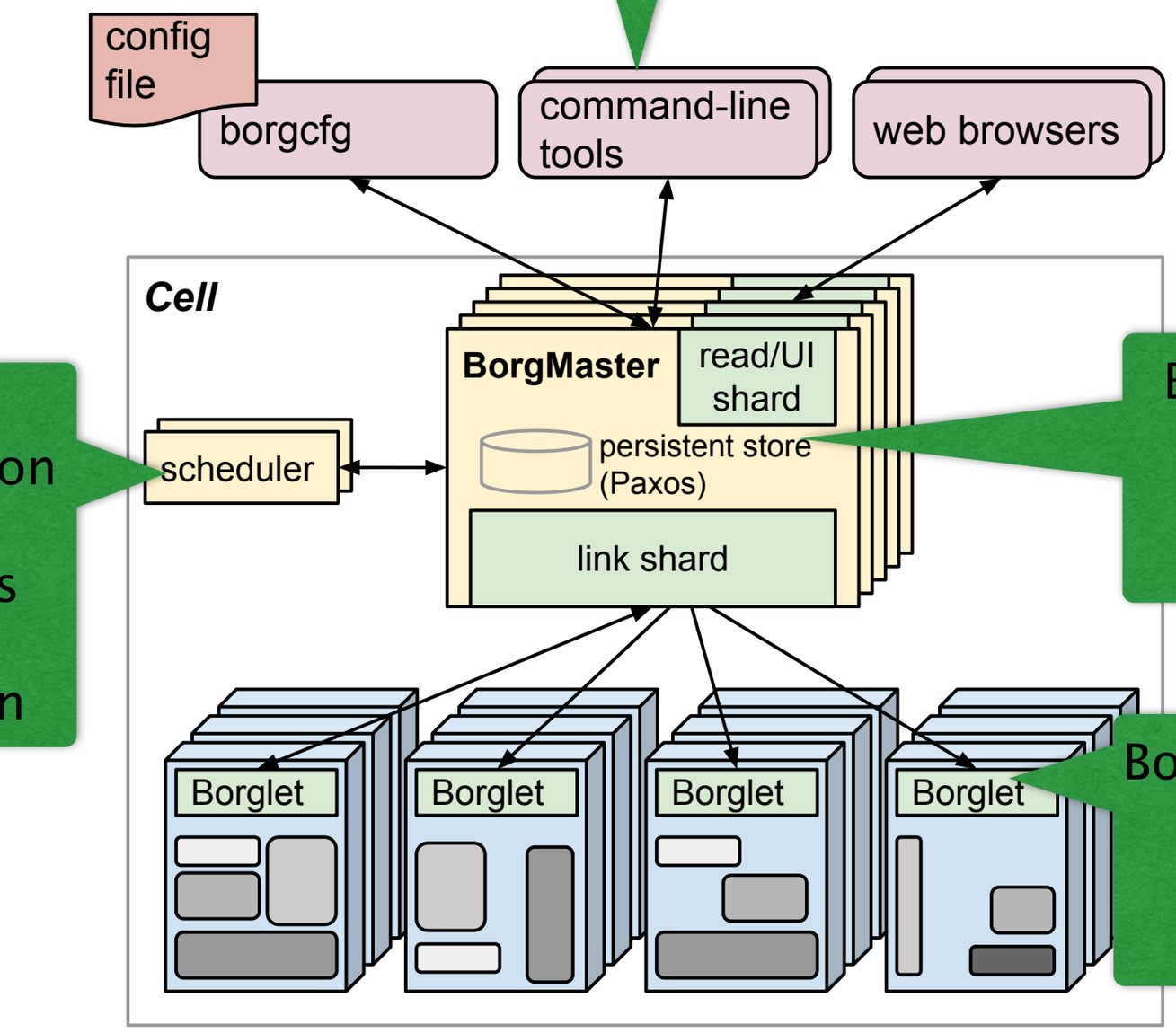
Scheduler
Decides on which machines task are placed on

BorgMaster
Central "brain": holds cluster state, replicated for reliability (Paxos)

Borglet
Per-machine agent, supervises local tasks and interacts with BorgMaster

Borg architecture

Front-ends
CLI/BCL/API: programmatic job submission, config description
Dashboards: monitoring of service SLAs (e.g., tail latency)
Sigma UI: introspection ("why are my tasks pending?")



Scheduler
Decides on which machines task are placed on

BorgMaster
Central "brain": holds cluster state, replicated for reliability (Paxos)

Borglet
Per-machine agent, supervises local tasks and interacts with BorgMaster

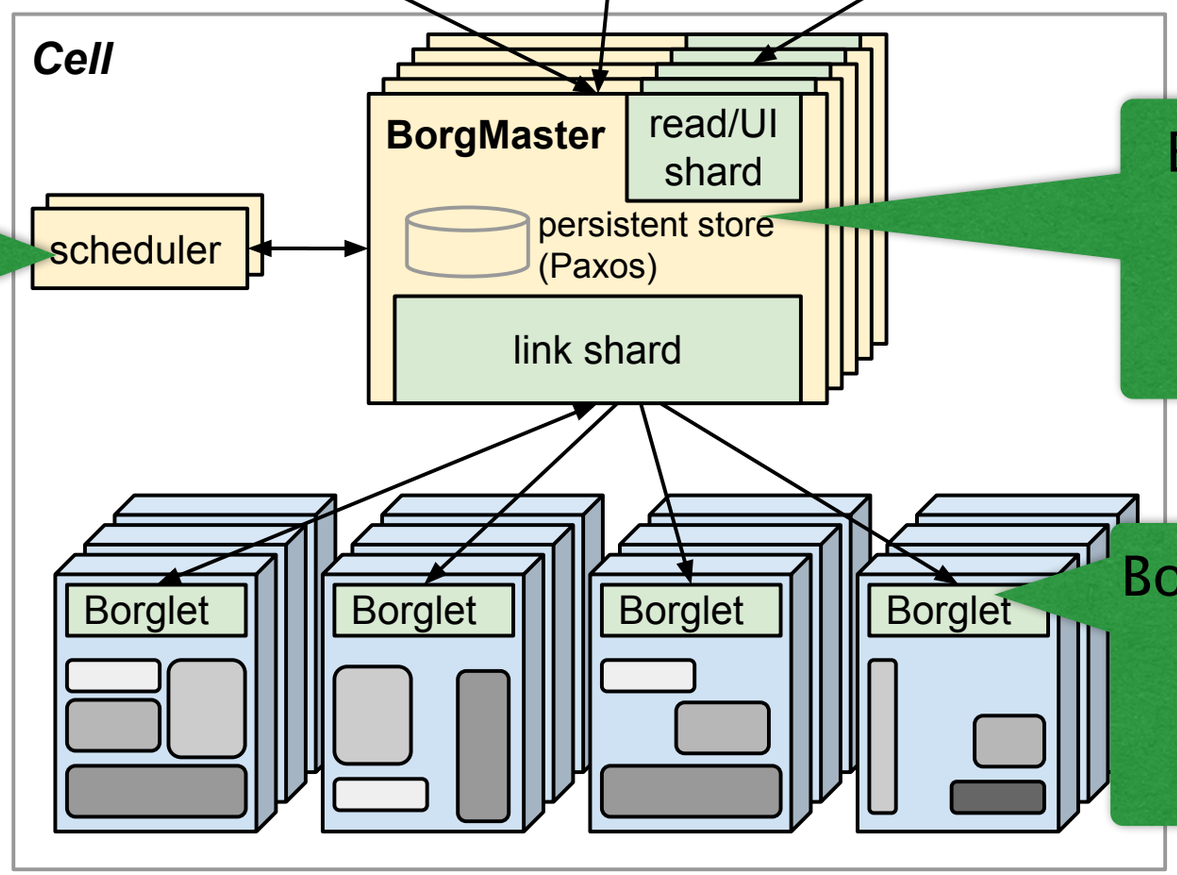
Job Specification (BCL)

```
job hello_world = {

  runtime = { cell = “ic” }                    // What cell should run it in?
  binary = ‘../hello_world_webserver’  // What program to run?
  args = { port = ‘%port%’ }

  requirements = {
    RAM = 100M
    disk = 100M
    CPU = 0.1
  }

  replicas = 10000
  …
}
```

Borg architecture

Front-ends
CLI/BCL/API: programmatic job submission, config description
Dashboards: monitoring of service SLAs (e.g., tail latency)

Clusters
A physical data center (building) holds multiple "clusters"
Each cluster is subdivided into logical "**cells**"
Borg manages a cell, users must themselves choose which cell to use
Geographically distributed, so jobs often replicated



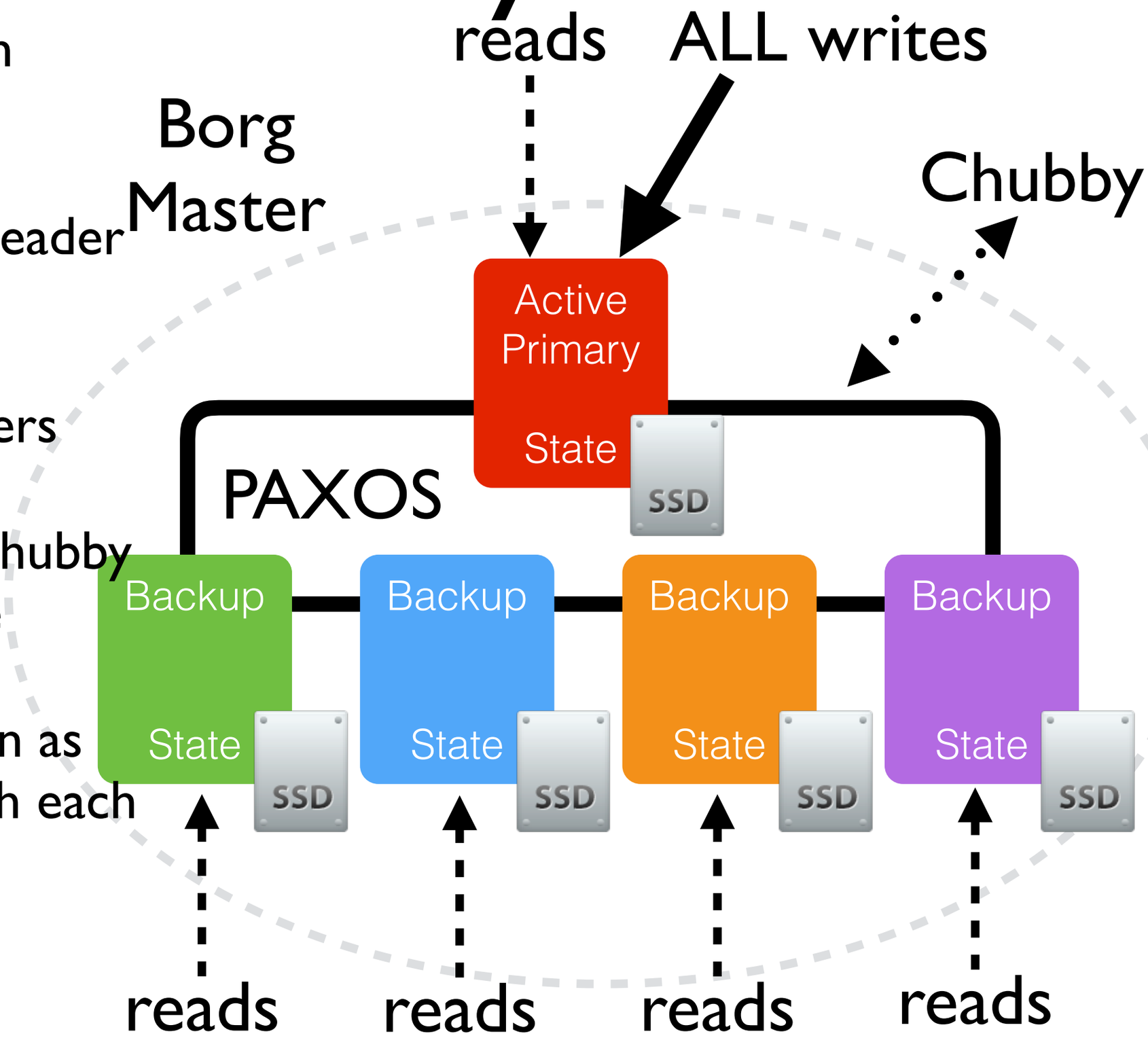
Scheduler
Decides on which machines task are placed on

BorgMaster
Central "brain": holds cluster state, replicated for reliability (Paxos)

Borglet
Per-machine agent, supervises local tasks and interacts with BorgMaster

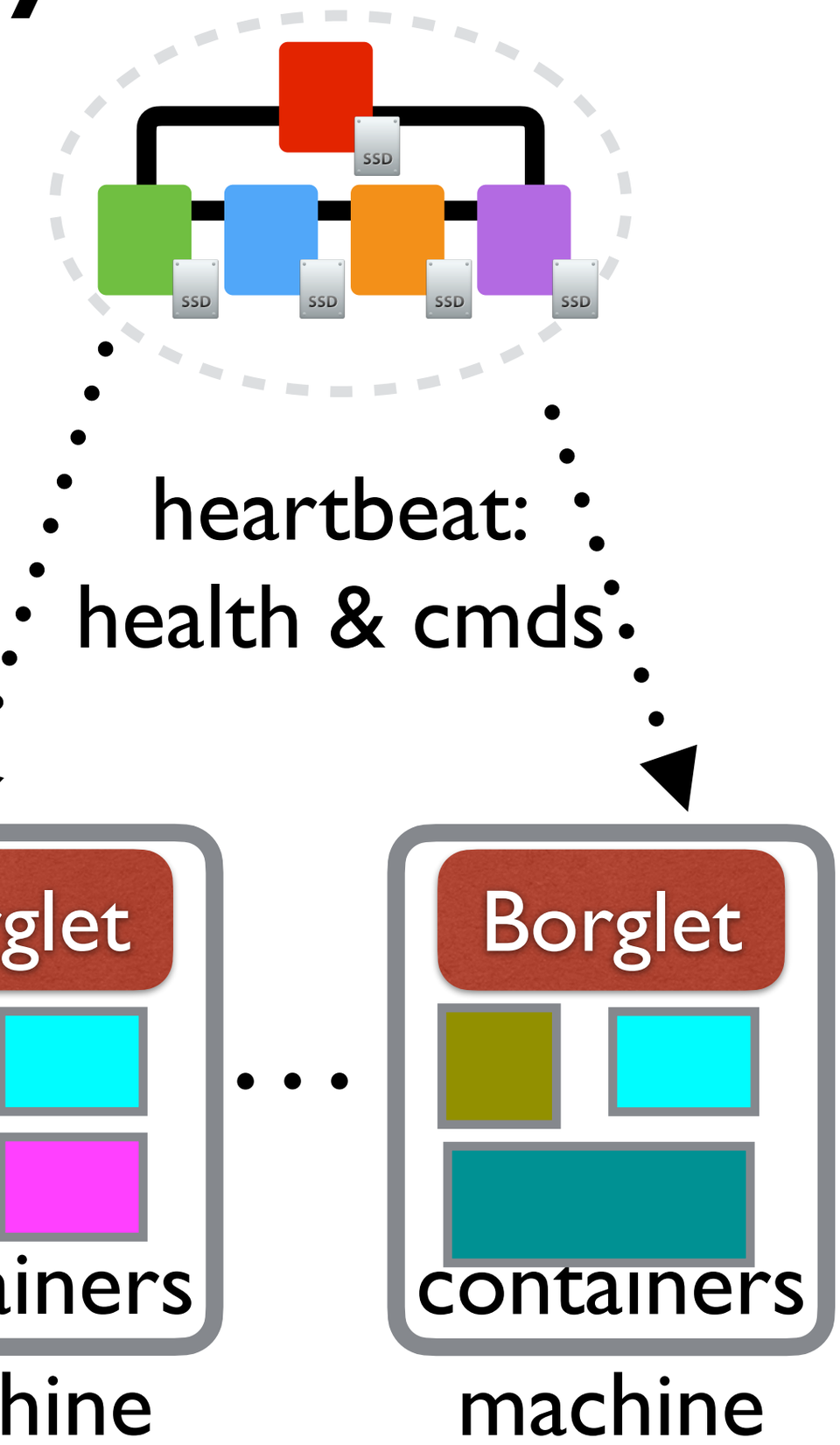
Reliability

- BORG Master RSM with persistent storage
  - One active Primary/Leader (via election)
  - Four Backups/Followers
  - Elections via Paxos/Chubby - take 10s to 1minute
  - After an outage: rejoin as backups and sync with each other
  - Checkpoints enable debugging and faster recovery



Reliability

- Borglet: local machine agent
  - status and directives through regular heart beats from master
- On loss of connection from master (BM failure/network partition) — carry on with what you are doing — on recovery then BM sort out any duplicates that might have been rescheduled



Workload

- Types
  - Services : serve user/job requests run indefinitely, generally latency-sensitive and high priority, FT matters
  - Batch : finite, terminate when done, not sensitive to short-term performance fluctuations
- Priority
  - Production: “monitoring” + “prod”: Guaranteed to run & always receives resourctes
  - Non-production: “batch” and “best-effort”: You get what you get

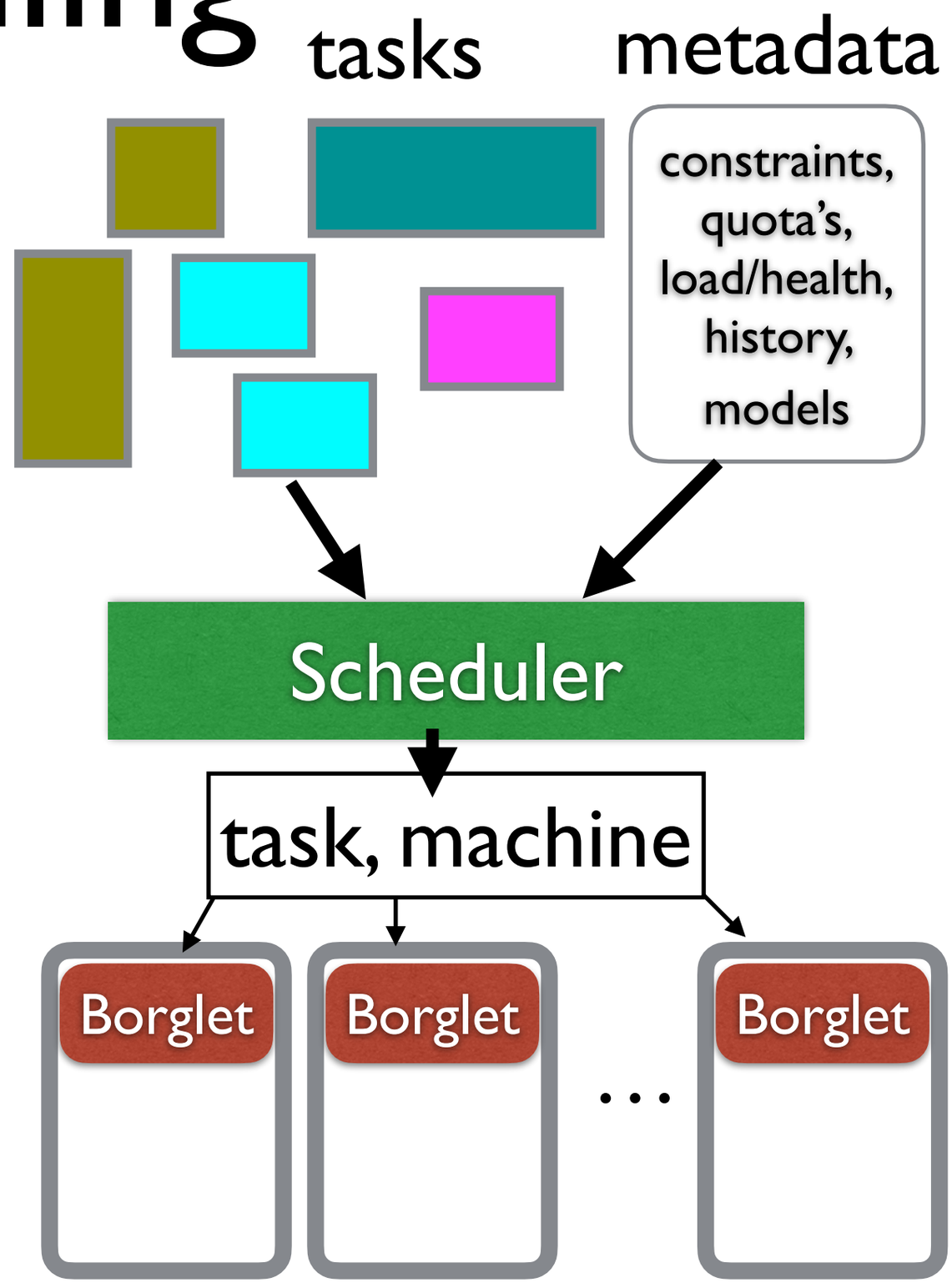
	Service	**Batch**
Production	BigTable, Gmail (FEs), model serving	GFS data migration
Batch		ML model training, data transformation
Best Effort	test of new service	intern MapReduce job, exploratory ML model training

Admission Control

- "Quota" — global notion of total capacity that must be "purchased" — human in the loop planning
  - Quota expensive to discourage overbuying
  - "low-quality" resources cheap — same hardware but preemptible and can be throttled
    - purchased for large time periods (eg monthly) — "Calendar" based. Mainly for production jobs
- Quota is NOT the per-task resource request!
  - Quota is per-user/per-team
  - Quota for medium-term capacity planning

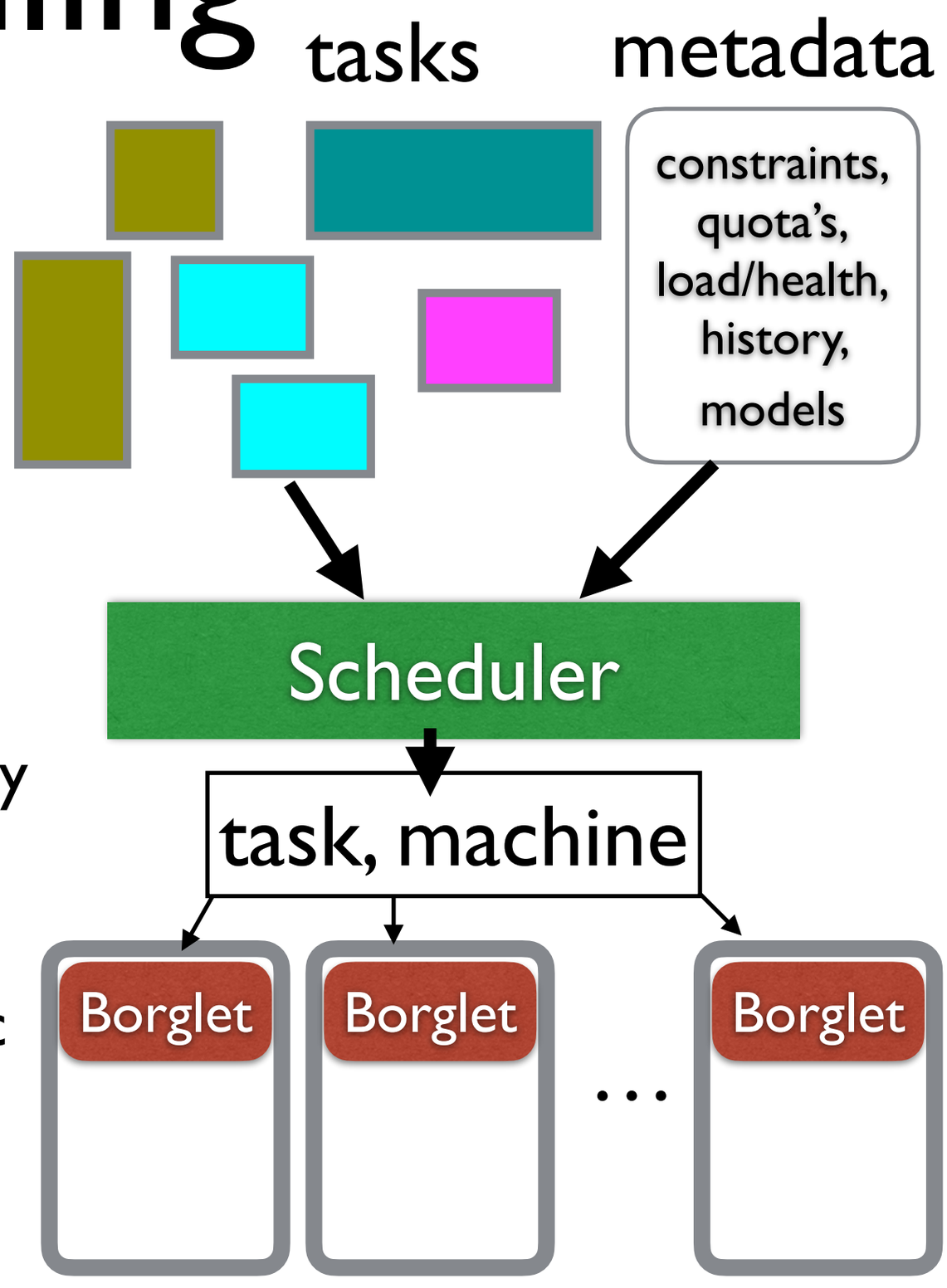
Scheduling

- The key question: "Where does each task go?" — mapping of containers to machines (Borglet)
- Impacts:
  - Fault Tolerance
  - Machine load
  - Time to start (wait time until place found)



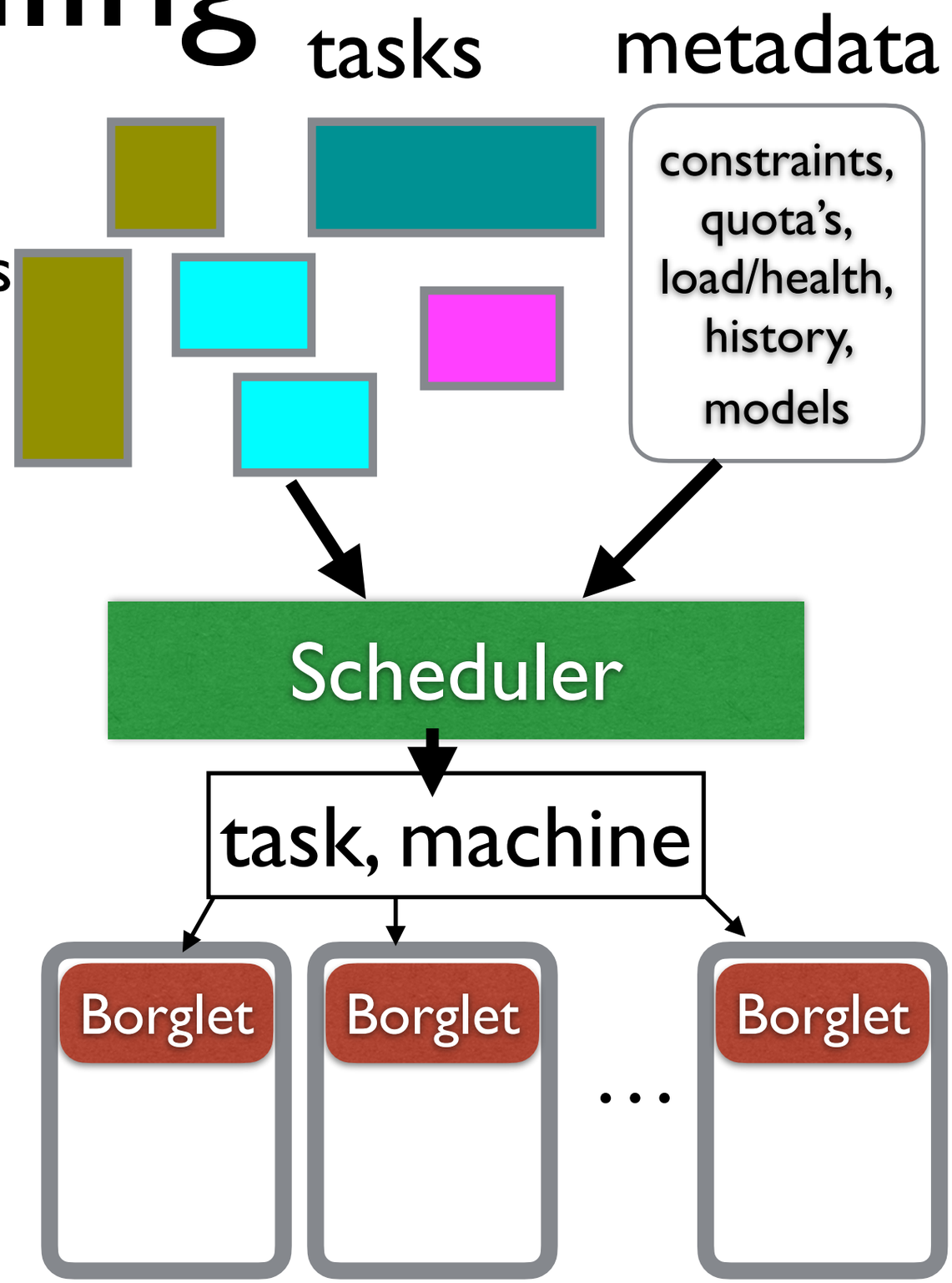
Scheduling

- Two main parts:
  - **<u>Feasibility checking:</u>** identify machines on which the task could run
    - Must have "enough available" resources in all dimensions for the task
      - "enough" depend on task priority
  - Placement constraints
    - hard: must be satisfied : eg. public ip, GPGPU
    - soft: preferences — eg. faster processor



Scheduling

- Two main parts:
  - **Scoring:** of the candidate machines which are the best choices?
    - use soft constraints
    - consider disruption: reduce the count and priority of preempted tasks
    - already has packages
    - spreads tasks across failure domains
    - Higher score = better packing



Utilization / Packing

- Why is this an issue?
  - Easy to waste resources + humans bad and estimates
- Reasons for waste
  - Users ask for more resources than they need
  - Load well balanced, but no room for large tasks (fragmentation)
  - Tight packing but no head room for expansion
  - Tasks' needs vary over time (e.g. diurnal service load)

Utilization / Packing

- Once the machine with highest score is picked for a task it is placed there
- This can result in `task compression` (for `compressible tasks`) or preemption:
  - From lowest to highest until the new task fits
  - Prempted tasks go back on pending queue

Cluster compaction metric

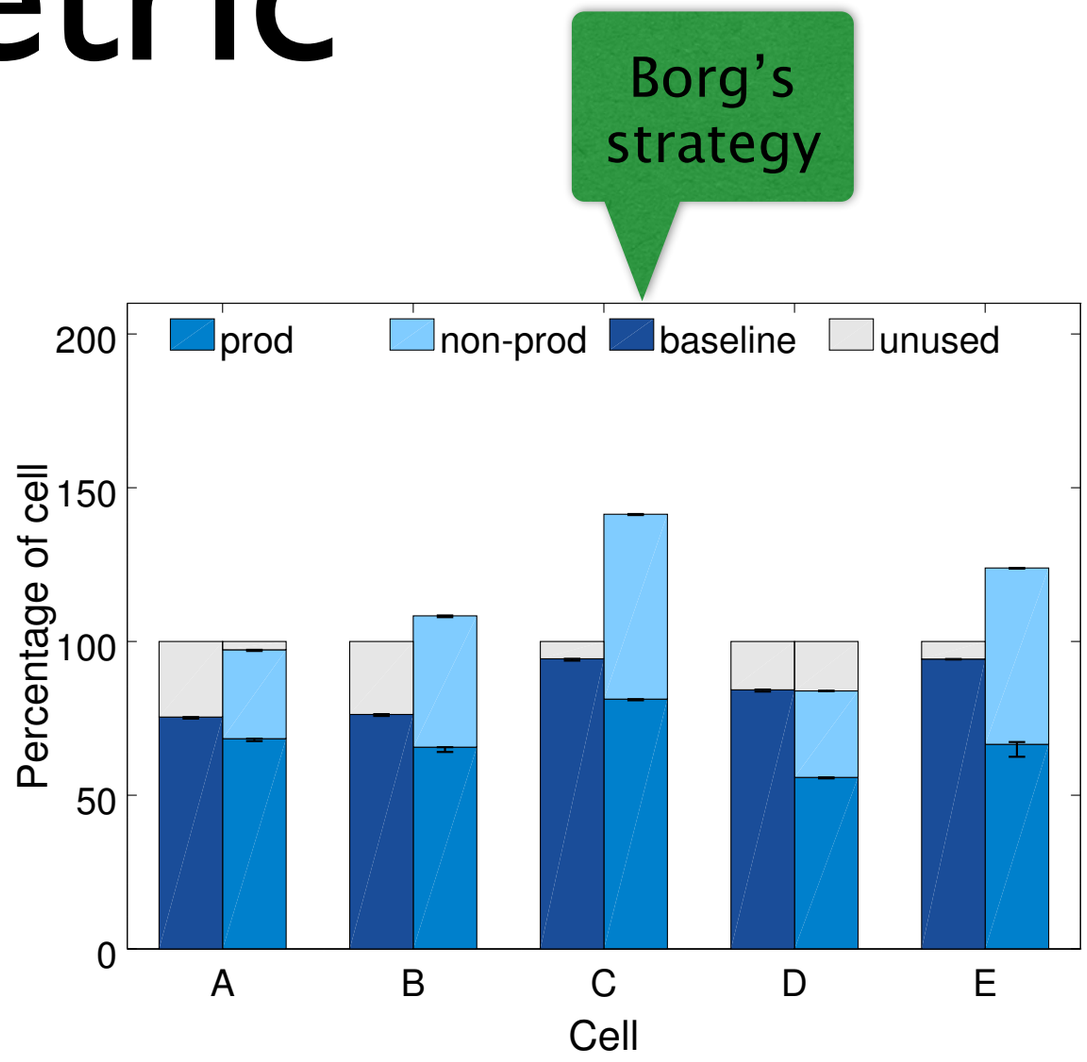
- How do we asses how good a job the scheduler did?
  - Could use cpu utilization, but says little about wastage
  - Could measure unusable “holes”, but usability of a hole is task-dependent (thus workload dependent)
  - Could artificially grow the workload, but hard to do realistically

Cluster compaction metric

- Key question: "How much smaller a cluster could I have gotten away with without running out of capacity — (many pending tasks)

- If the scheduler packs more tightly then a smaller cluster still works

- Of course a scheduler could do worse > 100% compared to baseline

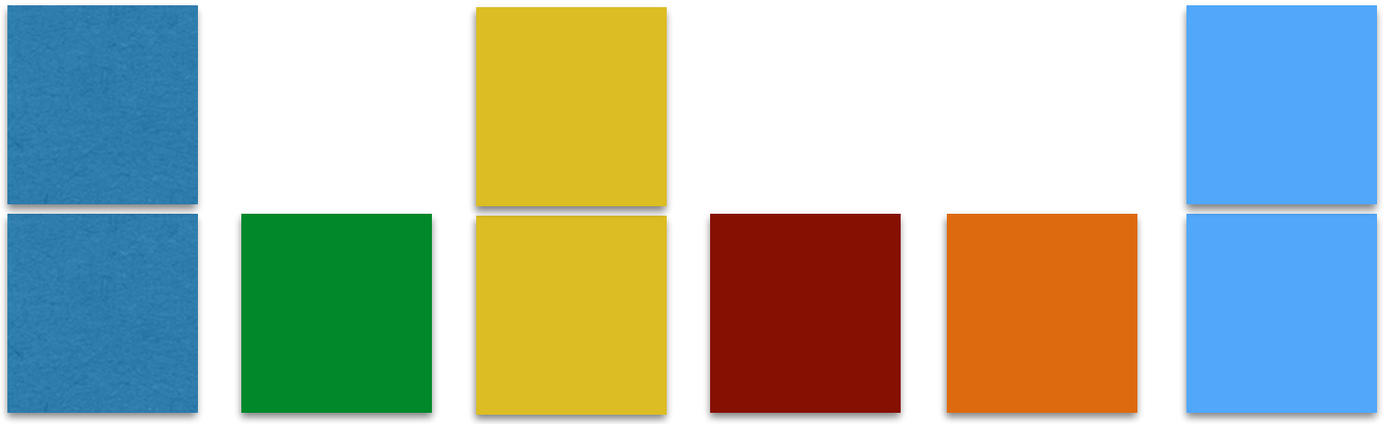


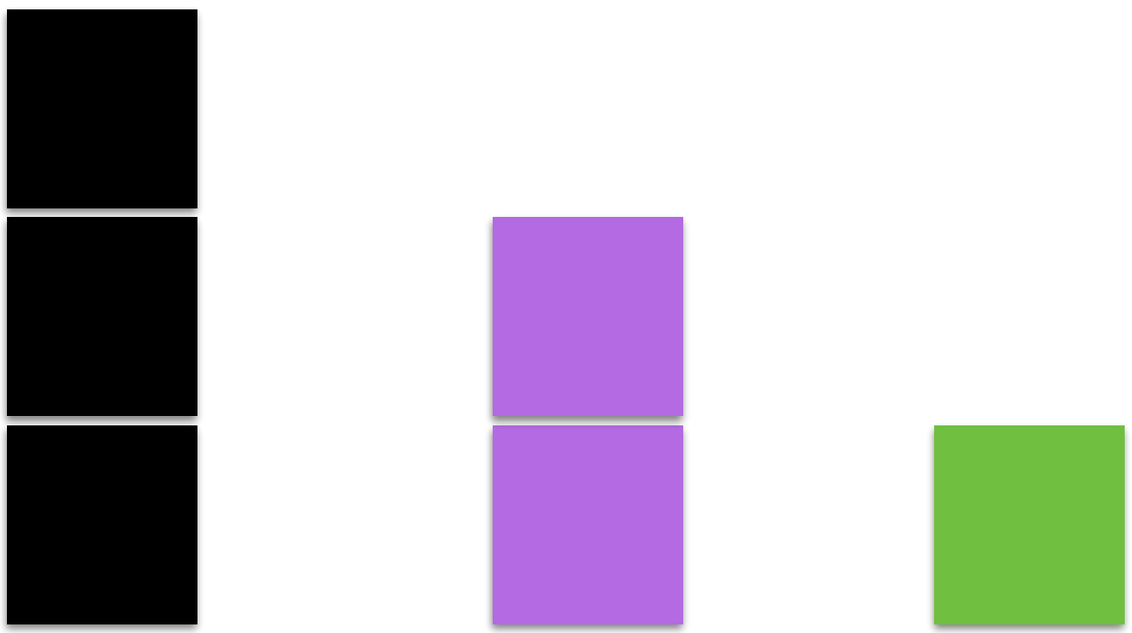
(a) *The left column for each cell shows the original size and the combined workload; the right one shows the segregated case.*

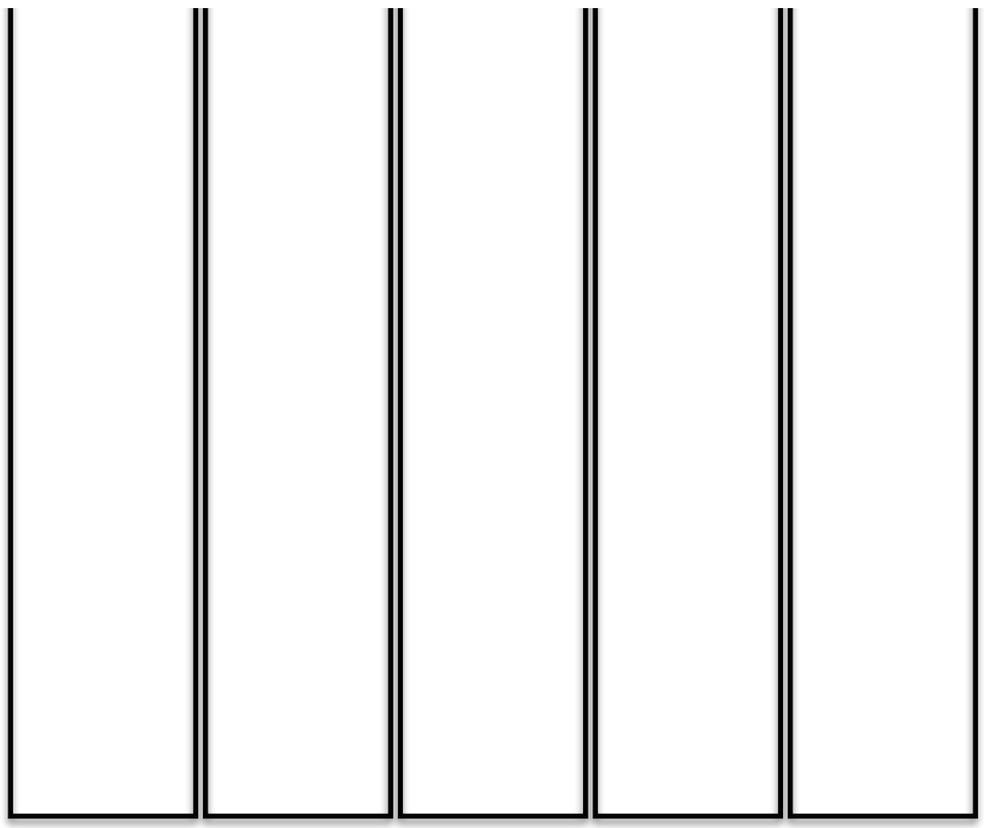
Repacking

Tasks
Requirements

Machines

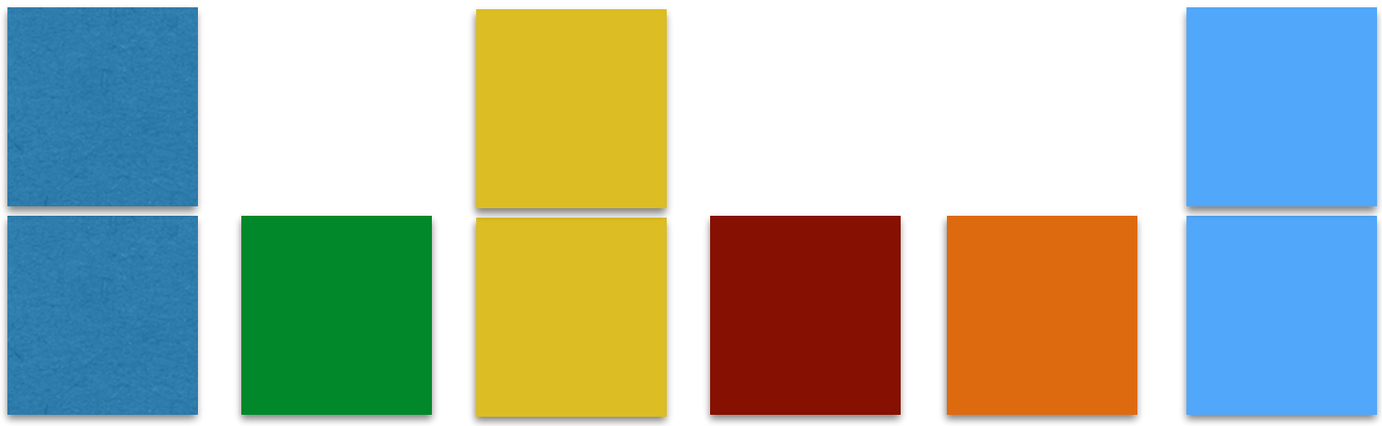


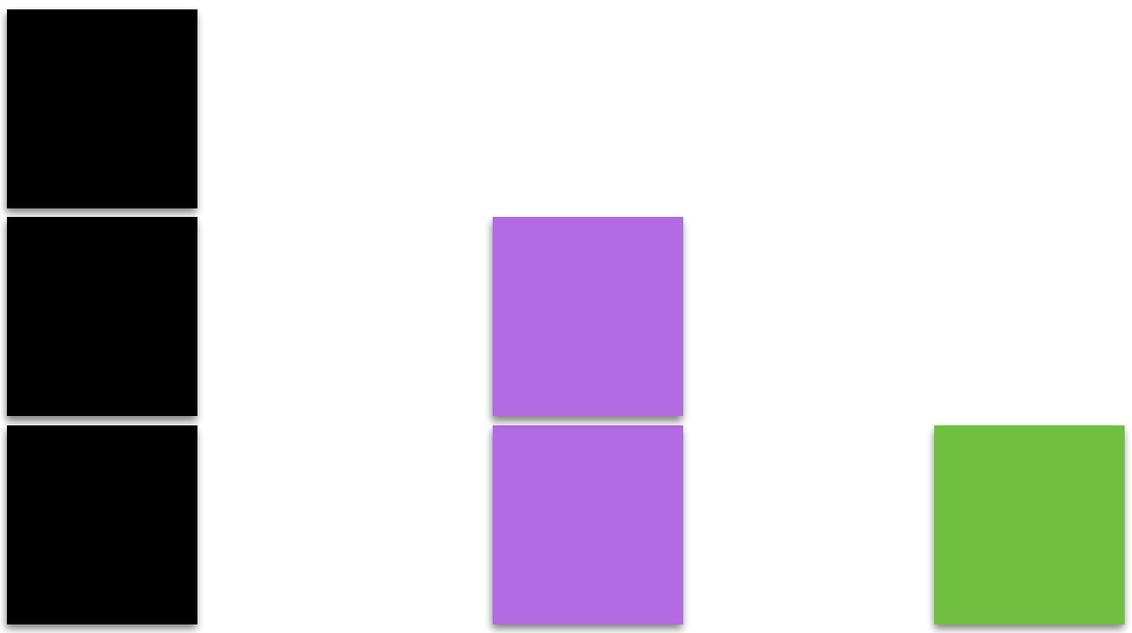




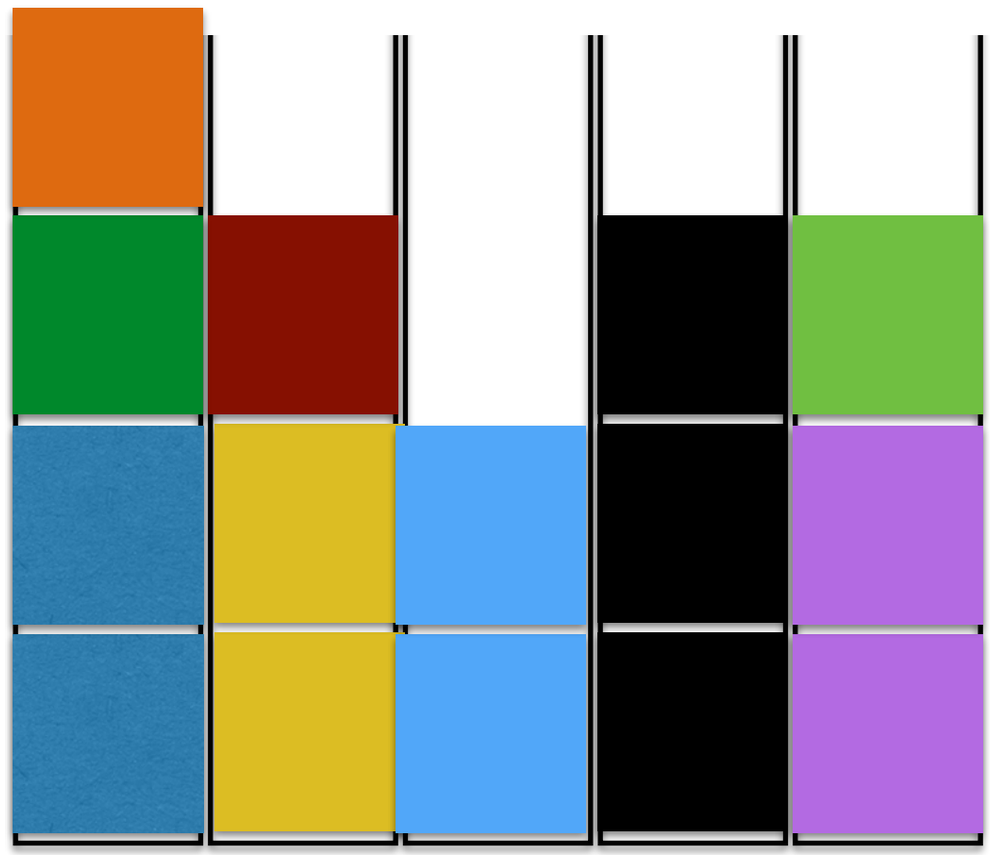
Repacking

Tasks
Requirements



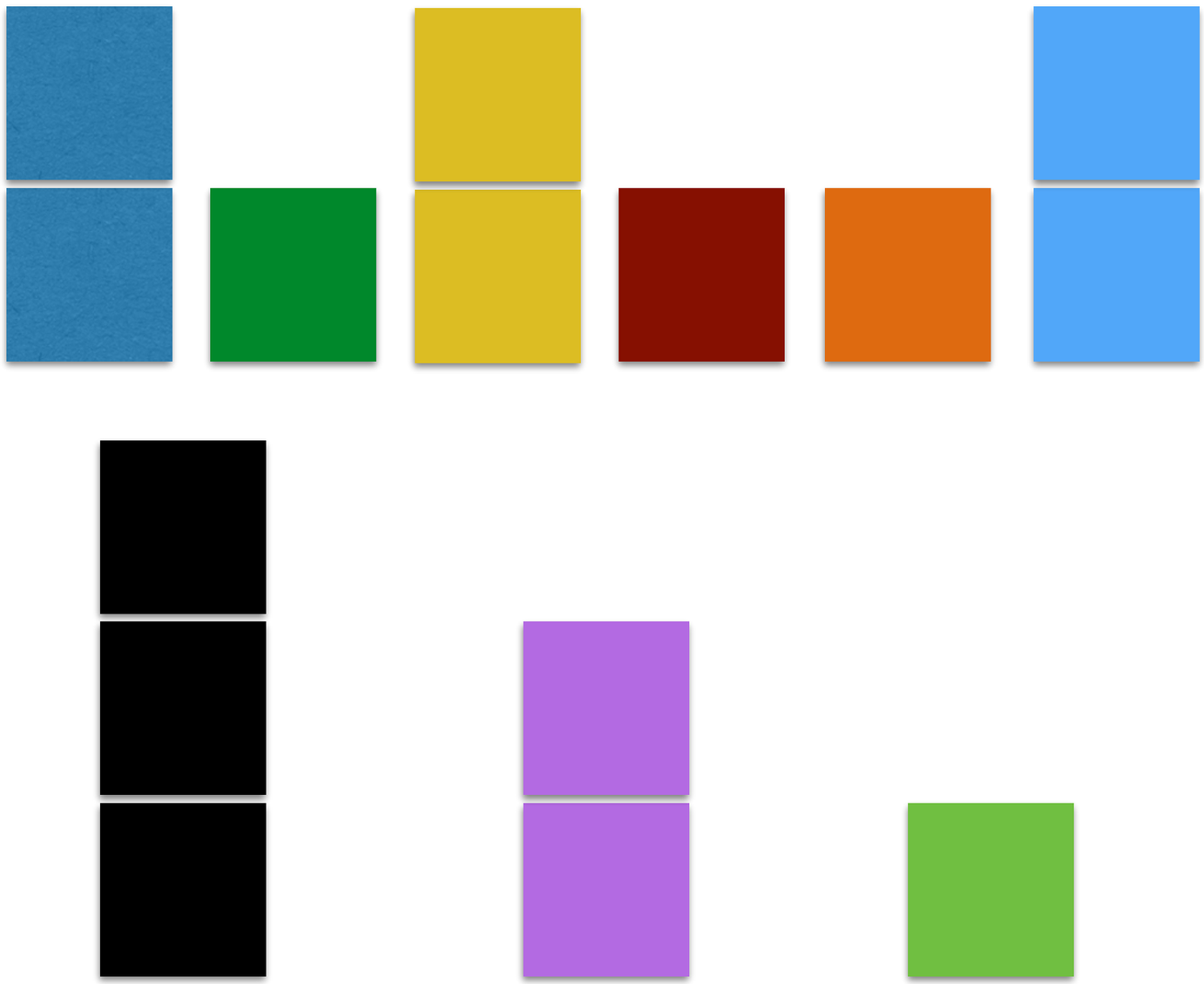


Packing
(in 5 machines)

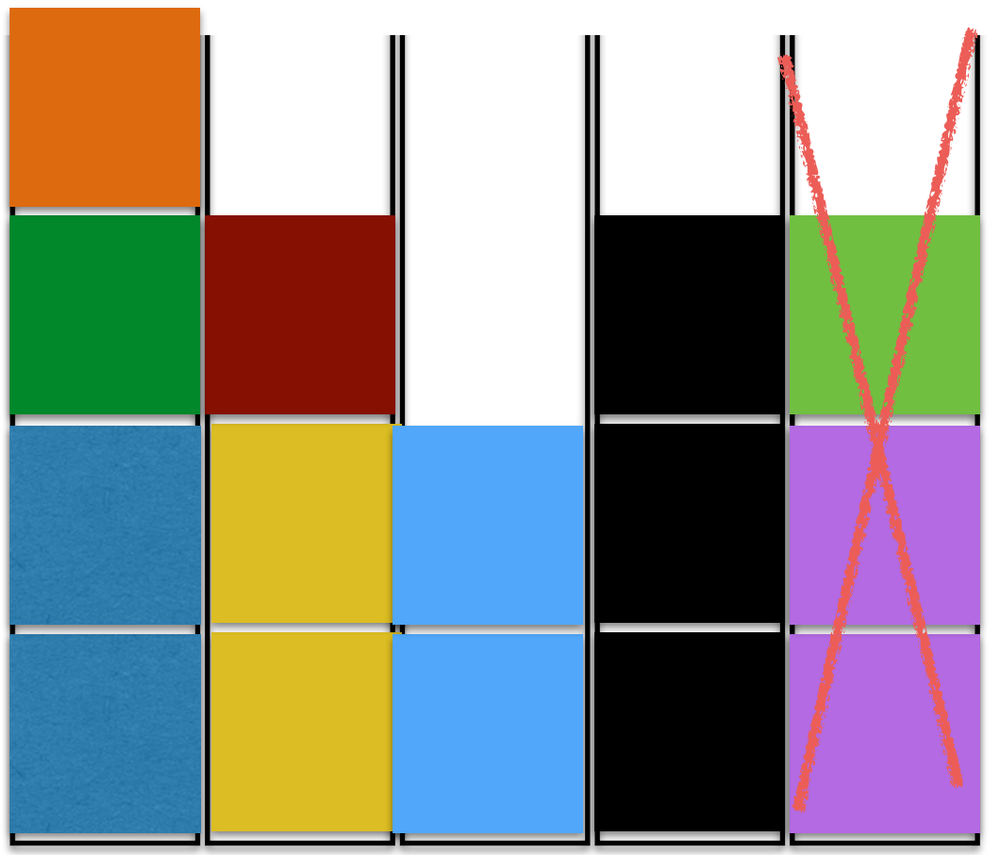


Repacking

Tasks
Requirements

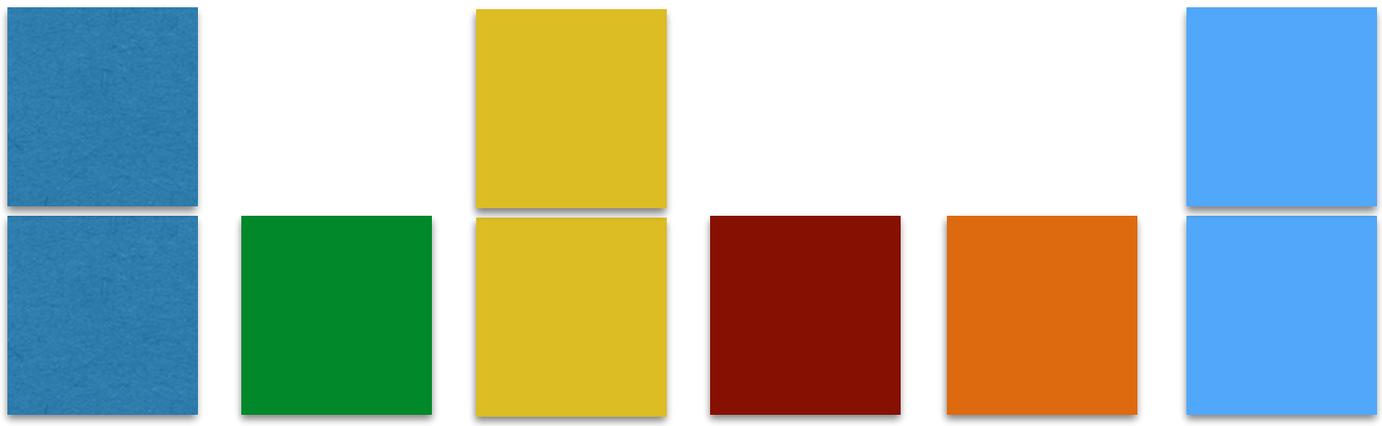


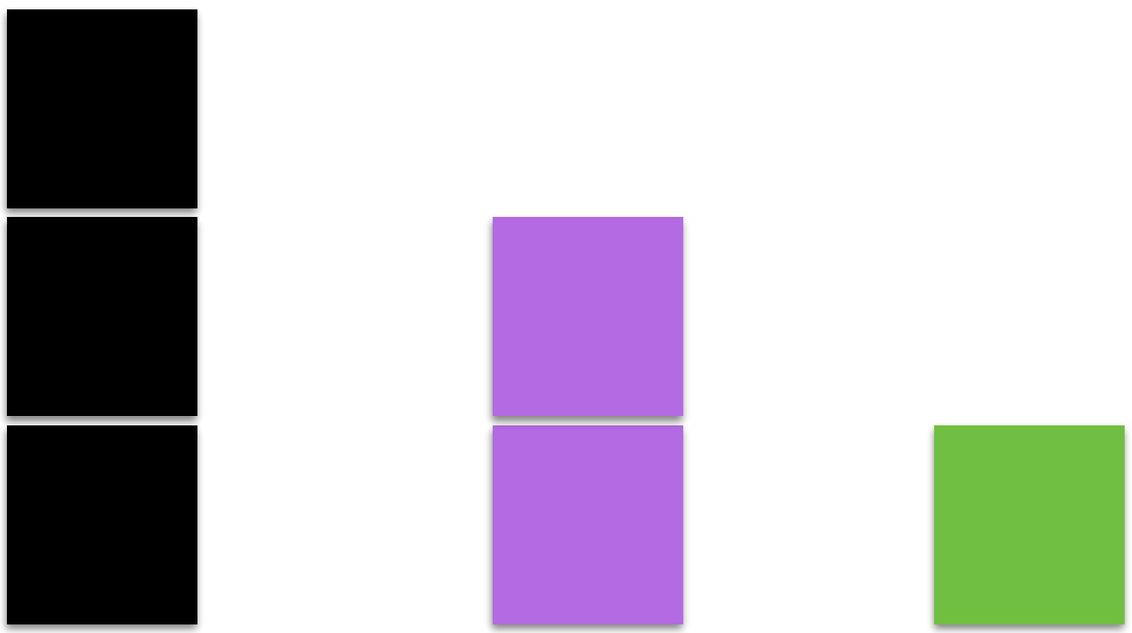
Remove one
machine



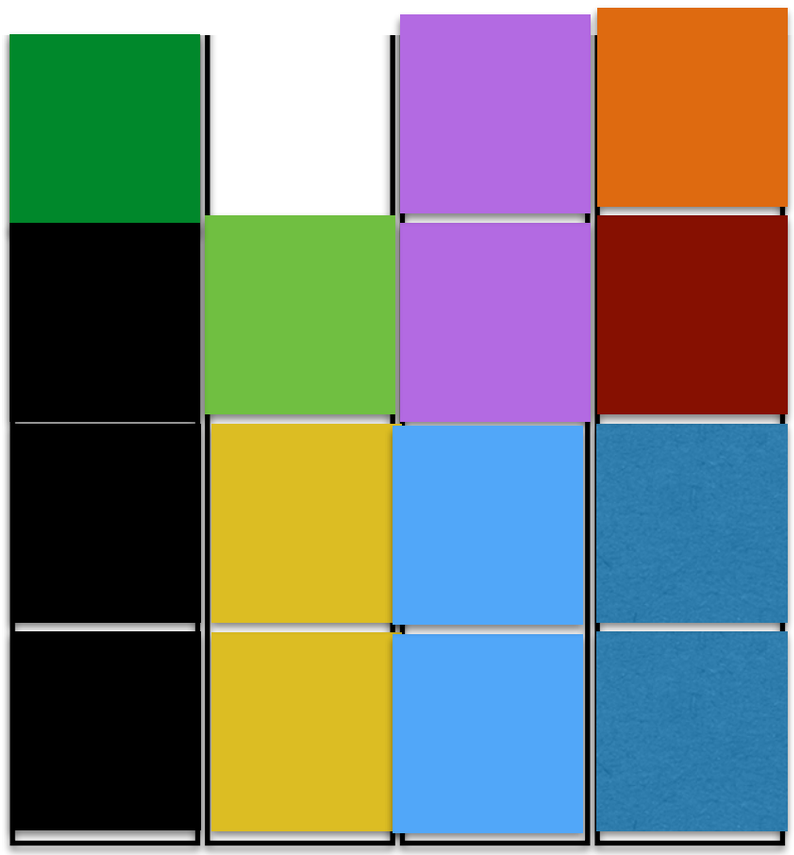
Repacking

Tasks
Requirements





Repacking
(in 4 machines)



Cluster compaction metric

Strategy A: 4 machines

Strategy B: 5 machines

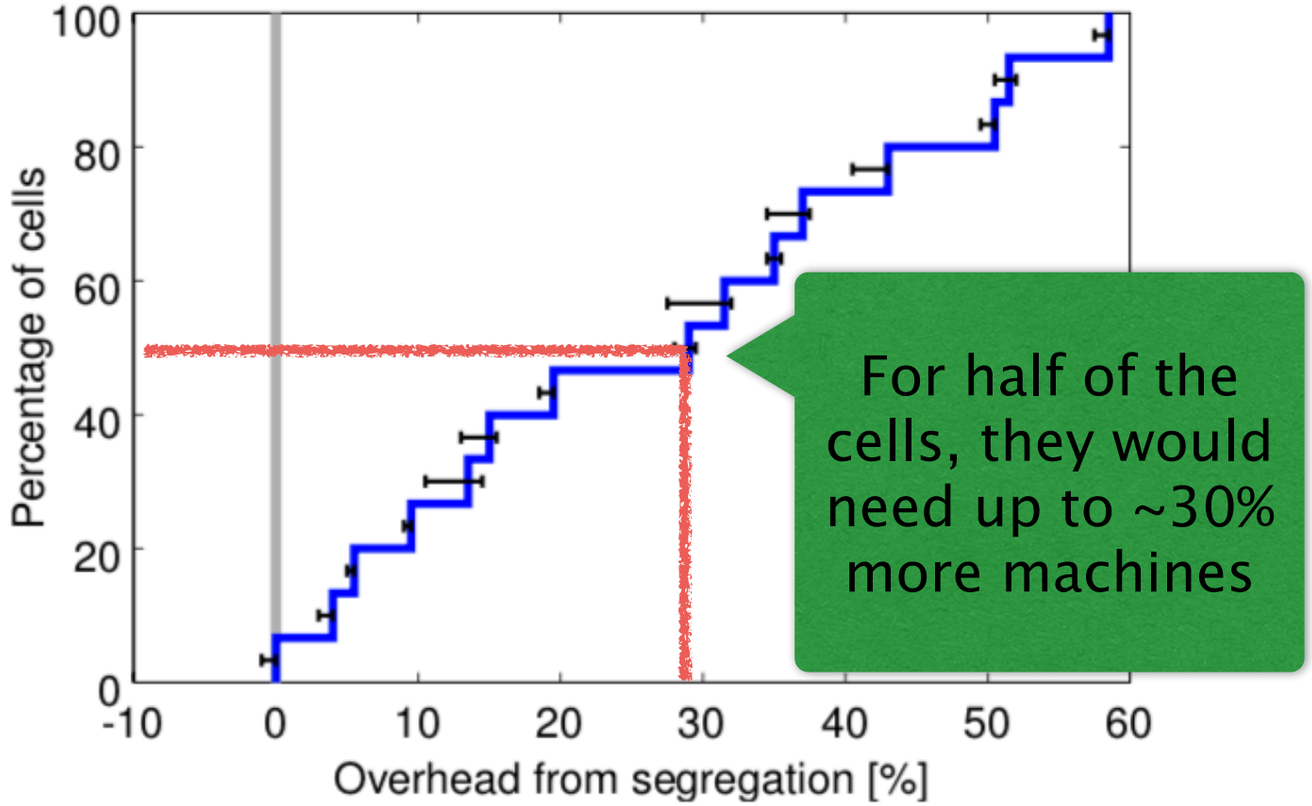
5-4 / 4 = 25% more machines if we use Strategy B

Cluster compaction metric

- Key question: "How much smaller a cluster could I have gotten away with without running out of capacity — (many pending tasks)

- If the scheduler packs more tightly then a smaller cluster still works

- Of course a scheduler could do worse > 100% compared to baseline



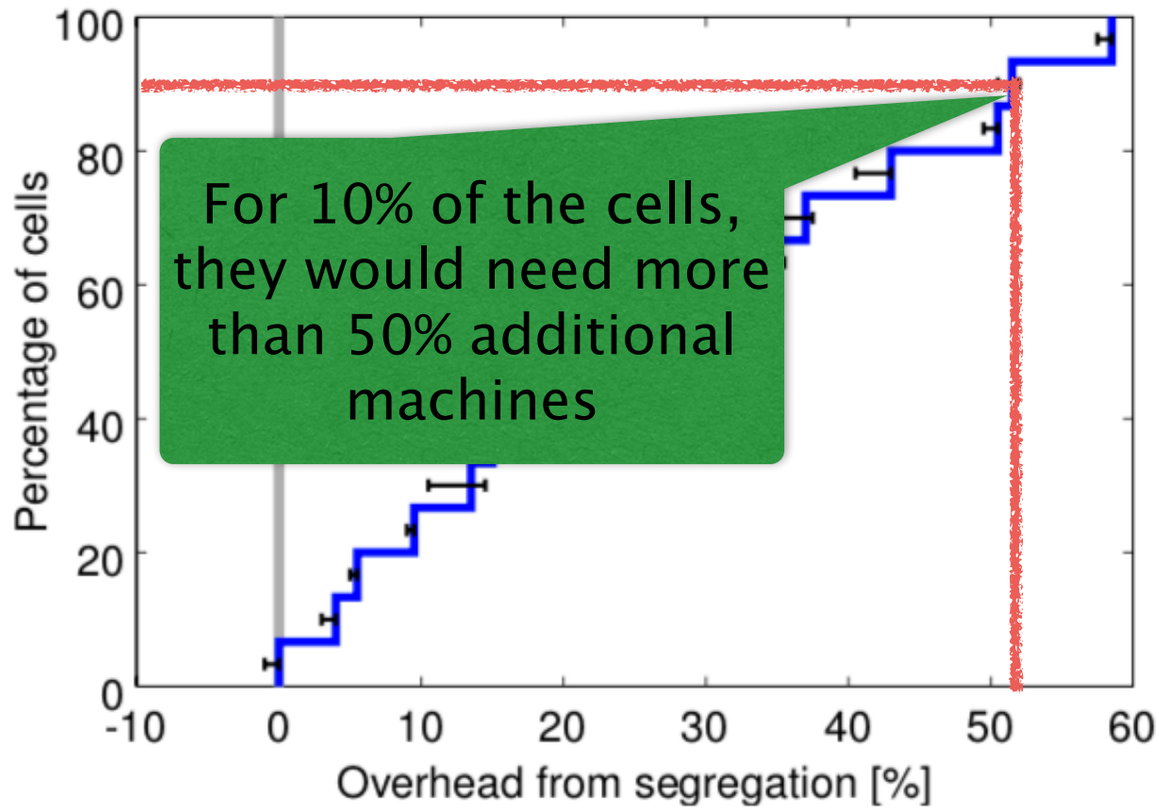
(b) *CDF of additional machines that would be needed if we segregated the workload of 15 representative cells.*

Cluster compaction metric

- Key question: "How much smaller a cluster could I have gotten away with without running out of capacity — (many pending tasks)

- If the scheduler packs more tightly then a smaller cluster still works

- Of course a scheduler could do worse > 100% compared to baseline



(b) *CDF of additional machines that would be needed if we segregated the workload of 15 representative cells.*

Idle resource avoidance

- Preemption
- Large, shared cells
- Fine-grain resource requests (e.g. “0.1 CPU”)
- Resource estimation
- Overcommit

Scaling

Many thousands of machines (>10k) in a cell and task arrival rates >10k per minute

- Non-trivial — lots of tricks
  - BorgMaster
    - distribute load over BorgMaster replicas (eg. link sharding of BorgLets)
    - threading (aggressive multi-core and io overlapping)
    - Careful co-design of master and scheduler to use readonly data

Scaling : 10K+ machines

- Non-trivial — lots of tricks
  - Scheduler
    - 2011 Borg cell trace: 150k+ tasks — Preemptions requires reconsidering all running tasks
      - can’t brute force it ~20k cycles per-task
        - Decompose/partition tasks into classes
        - Score caching
        - Relaxed randomization
          - Randomly sample candidate machines until enough are found — still can take a while for “picky tasks”