Containers, VMs, and Container Management Systems
VMs, Containers, and Container-Management Systems

Borg, Omega, and K8s
Borg, Omega, K8S

Borg
- Primary container management system within google
- Broad ecosystem of heterogeneous, ad-hoc systems built on top of Borg
- **Monolithic**, centralized master (Borgmaster) managing everything

Lots of complexity!

Omega
- Redesign of Borg to improve engineering of Borg ecosystem
- **Wanted a more consistent, principled architecture**
- Borgmaster functionality broken into separate “peer” components
- State stored in a centralized Paxos-based transaction oriented store (only centralized component)
- Exposes state store directly to trusted control-plane components

Manage complexity

Kubernetes
- Open source!
- Core is comprised of a shared persistent store, with components watching for changes to relevant objects
- State is accessed **EXCLUSIVELY** through a domain-specific REST API to support more diverse clients
- Focused on providing good experience to developers writing applications to run in a cluster

Enforce system-wide invariants, policies, data transforms

Not suitable for a public cloud, (potentially) inconsistent designs
Discussion Primer: Design Goals and Key Themes

● **Use case & design goals:**
  ○ Borg: **Internal-use** container management system
  ○ Omega: Borg but with better **software engineering practices**
  ○ Kubernetes: **Open source** container management software - emphasis on application developer experience

● **Centralization vs Componentization (Decoupling):**
  ○ Borg: Borgmaster is a **monolithic component** knowing semantics of every API operation
  ○ Omega: only has a centralized state store - all logic and semantics are pushed to store clients
    ■ Better scalability and consistency, less complexity
  ○ K8S: **componentized architecture** like Omega but enforces system-wide invariants, policies, and data transformation by funneling all store accesses through a centralized API server
Discussion
“The container has become the sole runnable entity supported by the google infrastructure.”
Why pick containers as the “sole runnable entity”?

- Raise the level of abstraction in data centers from managing machines to managing applications
  - Abstracting away specific details of machines and OSes for application development
  - Enables infrastructure teams to manage/upgrade infra with minimal application impact
  - Improves application monitoring and introspection (telemetry data tied to applications)
- Container provide various benefits in developing and running applications
  - Isolation
  - Utilization improvement
  - Dependency minimization and portability (decoupling)
    - Easy development, debugging, deployment, introspection
  - Lower overhead than VMs
- Are there any drawbacks to this approach?
How did resource isolation provided by containers enable higher utilization of machines at google?

● Colocate batch jobs with latency-sensitive, user-facing jobs.
  ○ User-facing jobs reserve more resources than they need (to handle load spikes and fail-over)
  ○ Batch jobs can reclaim unused resources (when not needed by user-facing jobs)

● Why do we care about latency-sensitive jobs and batch jobs?
  ○ Latency-sensitive jobs pay the bills (e.g. search) - allocated for peak load (resources oversubscribed)
  ○ Batch jobs - not as time-sensitive (e.g. analytics)
What do you think are some goals, non-goals, and constraints in the design of Borg, Omega, K8s?

- **Example goals, non-goals:**
  - Scalability, Flexibility, Performance (e.g. utilization), Efficiency, Consistency, Application development velocity/agility, Composability

- **Example constraints:**
  - Internal use vs Open Source

- **How did these influence design choices in Borg, Omega, and K8s? Is there anything you would do differently given the design requirements?**
Why do we care about consistency (of interfaces, system components, etc.) and what are the benefits of uniform APIs?

- Easier to learn system (generally systems are simpler)
- Easier to write *generic tools* and have *consistent user experiences*
- Uniform API → Control accesses to components
  - e.g. enforce system-wide invariants, policies, data transforms
- More generally how can we achieve consistency in our systems?
  - Uniform API
  - Decoupling - separation of concerns between API components → higher-level services all share the same common basic building blocks
  - Common design patterns
Page 12 mentions that Kubernetes is being extended to enable users to add their own APIs dynamically, alongside the core Kubernetes functionality.

- Trade-off between flexibility and consistency?
- Do you think this will create a large increase in complexity of systems?
- Recall that Borg had problems with a large collection of heterogeneous, ad-hoc systems...
- Recall that K8s is open source and is meant to support a diverse set of clients...
- Is this a good idea?
What is “control through choreography”? Why is it beneficial?

- Achieve a desired emergent behavior by **combining** the effects of **separate autonomous entities** that **collaborate**
  - Each entity has to manage its own (relatively) simple state space
  - Separation of concerns
- Centralized orchestration system (alternative design)
  - Easier to construct at first but becomes brittle and rigid over time, especially in the presence of unanticipated errors or state changes
  - Huge state space to consider, lots of complexity
What are some things to avoid? Why?

- Don’t make the container system manage port numbers
  - Assign unique port numbers (Borg) or IP addresses (K8s) to containers
- Don’t just number containers: give them labels (for grouping)
  - Indexing by numbers (Borg) vs labels (K8s)
  - Think about the scenarios where there are lots of containers
- Be careful with ownership
  - Tasks owned by jobs (Borg) vs separation of controllers and pods (K8s)
  - Think about debugging
- Don’t expose raw state (but we still need accesses to state)
  - Monolithic (Borg)
  - Componentized, state stored in centralized store, logic pushed into clients (Omega)
  - Centralized API server (K8s)
What are some open problems? Ideas?

- **Application configuration**
  - Commonly becomes the “catch-all” location for implementing things the container management system doesn’t do yet
  - (Solution?) want to maintain clear separation between computation and (configuration) data

- **Dependency management**
  - Services typically require other related services (e.g. monitoring, storage, CI/CD), how to automatically instantiate?
End!