Borg, Omega, and Kubernetes

LESSONS LEARNED FROM THREE CONTAINER-MANAGEMENT SYSTEMS OVER A DECADE

BRENDAN BURNS, BRIAN GRANT, DAVID OPPENHEIMER, ERIC BREWER, AND JOHN WILKES, GOOGLE INC.

Though widespread interest in software containers is a relatively recent phenomenon, at Google we have been managing Linux containers at scale for more than ten years and built three different container-management systems in that time. Each system was heavily influenced by its predecessors, even though they were developed for different reasons. This article describes the lessons we've learned from developing and operating them.

The first unified container-management system developed at Google was the system we internally call Borg.[7] It was built to manage both long-running services and batch jobs, which had previously been handled by two separate systems: Babysitter and the Global Work Queue. The latter's architecture strongly influenced Borg, but was focused on batch jobs; both predated Linux control groups. Borg shares machines between these two types of applications as a way of increasing resource utilization and thereby reducing costs. Such sharing was possible because container support in the Linux kernel was becoming available (indeed, Google contributed much of the container code to the Linux kernel), which enabled better isolation between latency-sensitive user-facing services and CPU-hungry batch processes.

As more and more applications were developed to run on top of Borg, our application and infrastructure teams developed a broad ecosystem of tools and services for it. These systems provided mechanisms for configuring and updating jobs; predicting resource requirements; dynamically pushing configuration files to running jobs; service discovery and load balancing; auto-scaling; machine-lifecycle management; quota management; and much more. The development of this ecosystem was driven by the needs of different teams inside Google, and the result was a somewhat heterogeneous, ad-hoc collection of systems that Borg's users had to configure and interact with, using several different configuration languages and processes. Borg remains the primary container-management system within Google because of its scale, breadth of features, and extreme robustness.

Omega,[6] an offspring of Borg, was driven by a desire to improve the software engineering of the Borg ecosystem. It applied many of the patterns that had proved successful in Borg, but was built from the ground up to have a more consistent, principled architecture. Omega stored the state of the cluster in a centralized Paxos-based transaction-oriented store that was accessed by the different parts of the cluster control plane (such as schedulers), using optimistic concurrency control to handle the occasional conflicts. This decoupling allowed the Borgmaster's functionality to be broken into separate components that acted as peers, rather than funneling every change through a monolithic, centralized master. Many of Omega's innovations (including

multiple schedulers) have since been folded into Borg.

The third container-management system developed at Google was Kubernetes.[4] It was conceived of and developed in a world where external developers were becoming interested in Linux containers, and Google had developed a growing business selling public-cloud infrastructure. Kubernetes is open source—a contrast to Borg and Omega, which were developed as purely Google-internal systems. Like Omega, Kubernetes has at its core a shared persistent store, with components watching for changes to relevant objects. In contrast to Omega, which exposes the store directly to trusted control-plane components, state in Kubernetes is accessed exclusively through a domain-specific REST API that applies higher-level versioning, validation, semantics, and policy, in support of a more diverse array of clients. More importantly, Kubernetes was developed with a stronger focus on the experience of developers writing applications that run in a cluster: its main design goal is to make it easy to deploy and manage complex distributed systems, while still benefiting from the improved utilization that containers enable.

This article describes some of the knowledge gained and lessons learned during Google's journey from Borg to Kubernetes.

CONTAINERS

Historically, the first containers just provided isolation of the root file system (via chroot), with FreeBSD jails extending

this to additional namespaces such as process IDs. Solaris subsequently pioneered and explored many enhancements. Linux control groups (cgroups) adopted many of these ideas, and development in this area continues today.

The resource isolation provided by containers has enabled Google to drive utilization significantly higher than industry norms. For example, Borg uses containers to co-locate batch jobs with latency-sensitive, user-facing jobs on the same physical machines. The user-facing jobs reserve more resources than they usually need—allowing them to handle load spikes and fail-over—and these mostly-unused resources can be reclaimed to run batch jobs. Containers provide the resource-management tools that make this possible, as well as robust kernel-level resource isolation to prevent the processes from interfering with one another. We achieved this by enhancing Linux containers concurrently with Borg's development. The isolation is not perfect, though: containers cannot prevent interference in resources that the operating-system kernel doesn't manage, such as level 3 processor caches and memory bandwidth, and containers need to be supported by an additional security layer (such as virtual machines) to protect against the kinds of malicious actors found in the cloud.

A modern container is more than just an isolation mechanism: it also includes an *image*—the files that make up the application that runs inside the container. Within Google, MPM (Midas Package Manager) is used to build and deploy container images. The same symbiotic relationship between the isolation mechanism and MPM packages can be found

between the Docker daemon and the Docker image registry. In the remainder of this article we use the word *container* to encompass both of these aspects: the runtime isolation and the image.

APPLICATION-ORIENTED INFRASTRUCTURE

Over time it became clear that the benefits of containerization go beyond merely enabling higher levels of utilization. Containerization transforms the data center from being machine-oriented to being application-oriented. This section discusses two examples:

- Containers encapsulate the application environment, abstracting away many details of machines and operating systems from the application developer and the deployment infrastructure.
- Because well-designed containers and container images are scoped to a single application, managing containers means managing applications rather than machines. This shift of management APIs from machine-oriented to application oriented dramatically improves application deployment and introspection.

Application environment

The original purpose of the cgroup, `chroot`, and namespace facilities in the kernel was to protect applications from noisy, nosey, and messy neighbors. Combining these with container images created an abstraction that also isolates applications from the (heterogeneous) operating systems

on which they run. This decoupling of image and OS makes it possible to provide the same deployment environment in both development and production, which, in turn, improves deployment reliability and speeds up development by reducing inconsistencies and friction.

The key to making this abstraction work is having a *hermetic* container image that can encapsulate almost all of an application's dependencies into a package that can be deployed into the container. If this is done correctly, the only local external dependencies will be on the Linux kernel system-call interface. While this limited interface dramatically improves the portability of images, it is not perfect: applications can still be exposed to churn in the OS interface, particularly in the wide surface area exposed by socket options, `/proc`, and arguments to `ioctl` calls. Our hope is that ongoing efforts such as the Open Container Initiative (https://www.opencontainers.org/) will further clarify the surface area of the container abstraction.

Nonetheless, the isolation and dependency minimization provided by containers have proved quite effective at Google, and the container has become the sole runnable entity supported by the Google infrastructure. One consequence is that Google has only a small number of OS versions deployed across its entire fleet of machines at any one time, and it needs only a small staff of people to maintain them and push out new versions.

There are many ways to achieve these hermetic images. In Borg, program binaries are statically linked at build time to known-good library versions hosted in the company-wide

repository.[5] Even so, the Borg container image is not quite as airtight as it could have been: applications share a so-called *base image* that is installed once on the machine rather than being packaged in each container. This base image contains utilities such as `tar` and the `libc` library, so upgrades to the base image can affect running applications and have occasionally been a significant source of trouble.

More modern container image formats such as Docker and ACI harden this abstraction further and get closer to the hermetic ideal by eliminating implicit host OS dependencies and requiring an explicit user command to share image data between containers.

Building management APIs around containers rather than machines shifts the "primary key" of the data center from machine to application.

Containers as the unit of management

Building management APIs around containers rather than machines shifts the "primary key" of the data center from machine to application. This has many benefits: (1) it relieves application developers and operations teams from worrying about specific details of machines and operating systems; (2) it provides the infrastructure team flexibility to roll out new hardware and upgrade operating systems with minimal impact on running applications and their developers; and (3) it ties telemetry collected by the management system (e.g., metrics such as CPU and memory usage) to applications rather than machines, which dramatically improves application monitoring and introspection, especially when scale-up, machine failures, or maintenance cause application instances to move.

Containers provide convenient points to register

generic APIs that enable the flow of information between the management system and an application without either knowing much about the particulars of the other's implementation. In Borg, this API is a series of HTTP endpoints attached to each container. For example, the `/healthz` endpoint reports application health to the orchestrator. When an unhealthy application is detected, it is automatically terminated and restarted. This self-healing is a key building block for reliable distributed systems. (Kubernetes offers similar functionality; the health check uses a user-specified HTTP endpoint or `exec` command that runs inside the container.)

Additional information can be provided by or for containers and displayed in various user interfaces. For example, Borg applications can provide a simple text status message that can be updated dynamically, and Kubernetes provides key-value *annotations* stored in each object's metadata that can be used to communicate application structure. Such annotations can be set by the container itself or other actors in the management system (e.g., the process rolling out an updated version of the container).

In the other direction, the container-management system can communicate information into the container such as resource limits, container metadata for propagation to logging and monitoring (e.g., user name, job name, identity), and notices that provide graceful-termination warnings in advance of node maintenance.

Containers can also provide application-oriented monitoring in other ways: for example, Linux kernel cgroups

provide resource-utilization data about the application, and these can be extended with custom metrics exported using HTTP APIs, as described earlier. This data enables the development of generic tools like an auto-scaler or cAdvisor[3] that can record and use metrics without understanding the specifics of each application. Because the container is the application, there is no need to (de)multiplex signals from multiple applications running inside a physical or virtual machine. This is simpler, more robust, and permits finer-grained reporting and control of metrics and logs. Compare this to having to **ssh** into a machine to run **top**. Though it is possible for developers to **ssh** into their containers, they rarely need to.

Monitoring is just one example. The application-oriented shift has ripple effects throughout the management infrastructure. Our load balancers don't balance traffic across machines; they balance across application instances. Logs are keyed by application, not machine, so they can easily be collected and aggregated across instances without pollution from multiple applications or system operations. We can detect application failures and more readily ascribe failure causes without having to disentangle them from machine-level signals. Fundamentally, because the identity of an instance being managed by the container manager lines up exactly with the identity of the instance expected by the application developer, it is easier to build, manage, and debug applications.

Finally, although so far we have focused on applications being 1:1 with containers, in reality we use nested containers

that are co-scheduled on the same machine: the outermost one provides a pool of resources; the inner ones provide deployment isolation. In Borg, the outermost container is called a resource allocation, or *alloc*; in Kubernetes, it is called a *pod*. Borg also allows top-level application containers to run outside allocs; this has been a source of much inconvenience, so Kubernetes regularizes things and always runs an application container inside a top-level pod, even if the pod contains a single container.

A common use pattern is for a pod to hold an instance of a complex application. The major part of the application sits in one of the child containers, and other child containers run supporting functions such as log rotation or click-log offloading to a distributed file system. Compared to combining the functionality into a single binary, this makes it easy to have different teams develop the distinct pieces of functionality, and it improves robustness (the offloading continues even if the main application gets wedged), composability (it's easy to add a new small support service, because it operates in the private execution environment provided by its own container), and fine-grained resource isolation (each runs in its own resources, so the logging system can't starve the main app, or vice versa).

Orchestration is the beginning, not the end

The original Borg system made it possible to run disparate workloads on shared machines to improve resource utilization. The rapid evolution of support services in the Borg ecosystem, however, showed that container

management *per se* was just the beginning of an environment for developing and managing reliable distributed systems. Many different systems have been built in, on, and around Borg to improve upon the basic container-management services that Borg provided. The following partial list gives an idea of their range and variety:

- Naming and service discovery (the Borg Name Service, or BNS).
- Master election, using Chubby.[2]
- Application-aware load balancing.
- Horizontal (number of instances) and vertical (size of an instance) autoscaling.
- Rollout tools that manage the careful deployment of new binaries and configuration data.
- Workflow tools (e.g., to allow running multijob analysis pipelines with interdependencies between the stages).
- Monitoring tools to gather information about containers, aggregate it, present it on dashboards, and use it to trigger alerts.

These services were built organically to solve problems that application teams experienced. The successful ones were picked up, adopted widely, and made other developers' lives easier. Unfortunately, these tools typically picked idiosyncratic APIs, conventions (such as file locations), and depth of Borg integration. An undesired side effect was to increase the complexity of deploying applications in the Borg ecosystem.

Kubernetes attempts to avert this increased complexity by adopting a consistent approach to its APIs. For example,

every Kubernetes object has three basic fields in its description: `ObjectMetadata, Specification` (or `Spec`), and `Status`.

The `Object Metadata` is the same for all objects in the system; it contains information such as the object's name, UID (unique identifier), an object version number (for optimistic concurrency control), and labels (key-value pairs, see below). The contents of `Spec` and `Status` vary by object type, but their concept does not: `Spec` is used to describe the *desired* state of the object, whereas `Status` provides read-only information about the *current* state of the object.

This uniform API provides many benefits. Learning the system is simpler: similar information applies to all objects, and writing generic tools that work across all objects is simpler, which in turn enables the development of a consistent user experience. Learning from Borg and Omega, Kubernetes is built from a set of composable building blocks that can readily be extended by its users. A common API and object-metadata structure makes that much easier. For example, the pod API is usable by people, internal Kubernetes components, and external automation tools. To further this consistency, Kubernetes is being extended to enable users to add their own APIs dynamically, alongside the core Kubernetes functionality.

Consistency is also achieved via decoupling in the Kubernetes API. Separation of concerns between API components means that higher-level services all share the same common basic building blocks. A good example of this is the separation between the Kubernetes replication

controller and its horizontal auto-scaling system. A replication controller ensures the existence of the desired number of pods for a given role (e.g., "front end"). The autoscaler, in turn, relies on this capability and simply adjusts the desired number of pods, without worrying about how those pods are created or deleted. The autoscaler implementation can focus on demand—and usage—predictions, and ignore the details of how to implement its decisions.

Decoupling ensures that multiple related but different components share a similar look and feel. For example, Kubernetes has three different forms of replicated pods:

- `ReplicationController`: run-forever replicated containers (e.g., web servers).
- `DaemonSet`: ensure a single instance on each node in the cluster (e.g., logging agents).
- `Job`: a run-to-completion controller that knows how to run a (possibly parallelized) batch job from start to finish.

Regardless of the differences in policy, all three of these controllers rely on the common pod object to specify the containers they wish to run.

Consistency is also achieved through common design patterns for different Kubernetes components. The idea of a *reconciliation controller loop* is shared throughout Borg, Omega, and Kubernetes to improve the resiliency of a system: it compares a desired state (e.g., how many pods should match a label-selector query) against the observed state (the number of such pods that it can find), and takes actions to converge the observed and desired states.

Because all action is based on observation rather than a state diagram, reconciliation loops are robust to failures and perturbations: when a controller fails or restarts it simply picks up where it left off.

The design of Kubernetes as a combination of microservices and small control loops is an example of control through *choreography*—achieving a desired emergent behavior by combining the effects of separate, autonomous entities that collaborate. This is a conscious design choice in contrast to a centralized *orchestration system*, which may be easier to construct at first but tends to become brittle and rigid over time, especially in the presence of unanticipated errors or state changes.

THINGS TO AVOID

While developing these systems we have learned almost as many things *not* to do as ideas that are worth doing. We present some of them here in the hopes that others can focus on making new mistakes, rather than repeating ours.

Don't make the container system manage port numbers

All containers running on a Borg machine share the host's IP address, so Borg assigns the containers unique port numbers as part of the scheduling process. A container will get a new port number when it moves to a new machine and (sometimes) when it is restarted on the same machine. This means that traditional networking services such as the DNS (Domain Name System) have to be replaced by home-

brew versions; service clients do not know the port number assigned to the service *a priori* and have to be told; port numbers cannot be embedded in URLs, requiring name-based redirection mechanisms; and tools that rely on simple IP addresses need to be rewritten to handle IP:port pairs.

Learning from our experiences with Borg, we decided that Kubernetes would allocate an IP address per pod, thus aligning network identity (IP address) with application identity. This makes it much easier to run off-the-shelf software on Kubernetes: applications are free to use static well-known ports (e.g., 80 for HTTP traffic), and existing, familiar tools can be used for things like network segmentation, bandwidth throttling, and management. All of the popular cloud platforms provide networking underlays that enable IP-per-pod; on bare metal, one can use an SDN (Software Defined Network) overlay or configure L3 routing to handle multiple IPs per machine.

Don't just number containers: give them labels

If you allow users to create containers easily, they tend to create lots of them, and soon need a way to group and organize them. Borg provides *jobs* to group identical *tasks* (its name for containers). A job is a compact vector of one or more identical tasks, indexed sequentially from zero. This provides a lot of power and is simple and straightforward, but we came to regret its rigidity over time. For example, when a task dies and has to be restarted on another machine, the same slot in the task vector has to do double duty: to identify the new copy and to point to the old one in case

it needs to be debugged. When tasks in the middle of the vector exit, the vector ends up with holes. The vector makes it very hard to support jobs that span multiple clusters in a layer above Borg. There are also insidious, unexpected interactions between Borg's job-update semantics (which typically restarts tasks in index order when doing rolling upgrades) and an application's use of the task index (e.g., to do sharding or partitioning of a dataset across the tasks): if the application uses range sharding based on the task index, Borg's restart policy can cause data unavailability, as it takes down adjacent tasks. Borg also provides no easy way to add application-relevant metadata to a job, such as role (e.g., "frontend"), or rollout status (e.g., "canary"), so people encode this information into job names that they decode using regular expressions.

In contrast, Kubernetes primarily uses *labels* to identify groups of containers. A label is a key/value pair that contains information that helps identify the object. A pod might have the labels `role=frontend` and `stage=production`, indicating that this container is serving as a production front-end instance. Labels can be dynamically added, removed, and modified by either automated tools or users, and different teams can manage their own labels largely independently. Sets of objects are defined by *label selectors* (e.g., `stage==production && role==frontend`). Sets can overlap, and an object can be in multiple sets, so labels are inherently more flexible than explicit lists of objects or simple static properties. Because a set is defined by a dynamic query, a new one can be created at any time. Label

selectors are *the* grouping mechanism in Kubernetes, and define the scope of all management operations that can span multiple entities.

Even in those circumstances where knowing the identity of a task in a set is helpful (e.g., for static role assignment and work-partitioning or sharding), appropriate per-pod labels can be used to reproduce the effect of task indexes, though it is the responsibility of the application (or some other management system external to Kubernetes) to provide such labeling. Labels and label selectors provide a general mechanism that gives the best of both worlds.

Be careful with ownership

In Borg, tasks do not exist independently from jobs. Creating a job creates its tasks; those tasks are forever associated with that particular job, and deleting the job deletes the tasks. This is convenient, but it has a major drawback: because there is only one grouping mechanism, it needs to handle all use cases. For example, a job has to store parameters that make sense only for service or batch jobs but not both, and users must develop workarounds when the job abstraction doesn't handle a use case (e.g., a `DaemonSet` that replicates a single pod to all nodes in the cluster).

In Kubernetes, pod-lifecycle management components such as replication controllers determine which pods they are responsible for using label selectors, so multiple controllers might think they have jurisdiction over a single pod. It is important to prevent such conflicts through appropriate configuration choices. But the flexibility of labels

has compensating advantages—for example, the separation of controllers and pods means that it is possible to "orphan" and "adopt" containers. Consider a load-balanced service that uses a label selector to identify the set of pods to send traffic to. If one of these pods starts misbehaving, that pod can be quarantined from serving requests by removing one or more of the labels that cause it to be targeted by the Kubernetes *service* load balancer. The pod is no longer serving traffic, but it will remain up and can be debugged *in situ*. In the meantime, the replication controller managing the pods that implements the service automatically creates a replacement pod for the misbehaving one.

A key difference between Borg, Omega, and Kubernetes is in their API architectures.

Don't expose raw state

A key difference between Borg, Omega, and Kubernetes is in their API architectures. The Borgmaster is a monolithic component that knows the semantics of *every* API operation. It contains the cluster management logic such as the state machines for jobs, tasks, and machines; and it runs the Paxos-based replicated storage system used to record the master's state. In contrast, Omega has no centralized component except the store, which simply holds passive state information and enforces optimistic concurrency control: all logic and semantics are pushed into the clients of the store, which directly read and write the store contents. In practice, every Omega component uses the same client-side library for the store, which does packing/unpacking of data structures, retries, and enforces semantic consistency.

Kubernetes picks a middle ground that provides the

flexibility and scalability of Omega's componentized architecture while enforcing system-wide invariants, policies, and data transformations. It does this by forcing all store accesses through a centralized API server that hides the details of the store implementation and provides services for object validation, defaulting, and versioning. As in Omega, the client components are decoupled from one another and can evolve or be replaced independently (which is especially important in the open-source environment), but the centralization makes it easy to enforce common semantics, invariants, and policies.

SOME OPEN, HARD PROBLEMS

Even with years of container-management experience, we feel there are a number of problems that we still don't have good answers for. This section describes a couple of particularly knotty ones, in the hope of fostering discussion and solutions.

Configuration

Of all the problems we have confronted, the ones over which the most brainpower, ink, and code have been spilled are related to managing *configurations*—the set of values supplied to applications, rather than hard-coded into them. In truth, we could have devoted this entire article to the subject and still have had more to say. What follows are a few highlights.

First, application configuration becomes the catch-all location for implementing all of the things that the

container-management system doesn't (yet) do. Over the history of Borg this has included:

- Boilerplate reduction (e.g., defaulting task-restart policies appropriate to the workload, such as service or batch jobs).
- Adjusting and validating application parameters and command-line flags.
- Implementing workarounds for missing API abstractions such as package (image) management.
- Libraries of configuration templates for applications.
- Release-management tools.
- Image version specification.

To cope with these kinds of requirements, configuration-management systems tend to invent a domain-specific configuration language that (eventually) becomes Turing complete, starting from the desire to perform computation on the data in the configuration (e.g., to adjust the amount of memory to give a server as a function of the number of shards in the service). The result is the kind of inscrutable "configuration is code" that people were trying to avoid by eliminating hard-coded parameters in the application's source code. It doesn't reduce operational complexity or make the configurations easier to debug or change; it just moves the computations from a real programming language to a domain-specific one, which typically has weaker development tools such as debuggers and unit test frameworks.

We believe the most effective approach is to accept this need, embrace the inevitability of programmatic configuration, and maintain a clean separation between computation and data. The language to represent the data

should be a simple, data-only format such as JSON or YAML, and programmatic modification of this data should be done in a real programming language, where there are well-understood semantics, as well as good tooling. Interestingly, this same separation of computation and data can be seen in front-end development with frameworks such as Angular that maintain a crisp separation between the worlds of markup (data) and JavaScript (computation).

Dependency management

Standing up a service typically also means standing up a series of related services (monitoring, storage, Continuous Integration / Continuous Deployment (CI/CD), etc). If an application has dependencies on other applications, wouldn't it be nice if those dependencies (and any transitive dependencies they may have) were automatically instantiated by the cluster-management system?

To complicate things, instantiating the dependencies is rarely as simple as just starting a new copy—for example, it may require registering as a consumer of an existing service (e.g., Bigtable as a service) and passing authentication, authorization, and billing information across those transitive dependencies. Almost no system, however, captures, maintains, or exposes this kind of dependency information, so automating even common cases at the infrastructure level is nearly impossible. Turning up a new application remains complicated for the user, making it harder for developers to build new services, and often results in the most recent best

practices not being followed, which affects the reliability of the resulting service.

A standard problem is that it is hard to keep dependency information up to date if it is provided manually, and at the same time attempts to determine it automatically (e.g., by tracing accesses) fail to capture the semantic information needed to understand the result. (Did that access have to go to *that* instance, or would any instance have sufficed?) One possible way to make progress is to require that an application enumerate the services on which it depends, and have the infrastructure refuse to allow access to any others. (We do this for compiler imports in our build system.[1]) The incentive would be enabling the infrastructure to do useful things in return, such as automatic setup, authentication, and connectivity.

Unfortunately, the perceived complexity of systems that express, analyze, and use system dependencies has been too high, and so they haven't yet been added to a mainstream container-management system. We still hope that Kubernetes might be a platform on which such tools can be built, but doing so remains an open challenge.

CONCLUSIONS

A decade's worth of experience building container-management systems has taught us much, and we have embedded many of those lessons into Kubernetes, Google's most recent container-management system. Its goals are to build on the capabilities of containers to provide significant gains

in programmer productivity and ease of both manual and automated system management. We hope you'll join us in extending and improving it.

References

1. Bazel: {fast, correct}—choose two; http://bazel.io.
2. Burrows, M. 2006. The Chubby lock service for loosely coupled distributed systems. Symposium on Operating System Design and Implementation (OSDI), Seattle, WA.
3. cAdvisor; https://github.com/google/cadvisor.
4. Kubernetes; http://kubernetes.io/.
5. Metz, C. 2015. Google is 2 billion lines of code—and it's all in one place. *Wired* (September); http://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/.
6. Schwarzkopf, M., Konwinski, A., Abd-el-Malek, M., Wilkes, J. 2013. Omega: flexible, scalable schedulers for large compute clusters. European Conference on Computer Systems (EuroSys), Prague, Czech Republic.
7. Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., Wilkes, J. 2015. Large-scale cluster management at Google with Borg. European Conference on Computer Systems (EuroSys), Bordeaux, France.

LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

Brendan Burns *(@brendandburns) is a software engineer at Google, where he co-founded the Kubernetes project. He received his Ph.D. from the University of Massachusetts Amherst in 2007. Prior to working on Kubernetes and cloud,*

he worked on low-latency indexing for Google's web-search infrastructure.

Brian Grant *is a software engineer at Google. He was previously a technical lead of Borg and founder of the Omega project and is now design lead of Kubernetes.*

David Oppenheimer *is a software engineer at Google and a tech lead on the Kubernetes project. He received a PhD from UC Berkeley in 2005 and joined Google in 2007, where he was a tech lead on the Borg and Omega cluster-management systems prior to Kubernetes.*

Eric Brewer *is VP Infrastructure at Google and a professor at UC Berkeley, where he pioneered scalable servers and elastic infrastructure.*

John Wilkes *has been working on cluster management and infrastructure services at Google since 2008. Before that, he spent time at HP Labs, becoming an HP and ACM Fellow in 2002. He is interested in far too many aspects of distributed systems, but a recurring theme has been technologies that allow systems to manage themselves. In his spare time he continues, stubbornly, trying to learn how to blow glass.*

Copyright © 2016 by the ACM. All rights reserved.