O'REILLY®

Case Studies in Infrastructure Change Management

How Google Rebuilds the Jet While Flying It

Wendy Look & Mark Dallman

REPORT

Google Cloud

Want to know more about SRE?

To learn more, visit google.com/sre

Case Studies in Infrastructure Change Management

How Google Rebuilds the Jet While Flying It

Wendy Look and Mark Dallman

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Case Studies in Infrastructure Change Management
by Wendy Look and Mark Dallman

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Virginia Wilson and John Devins
Production Editor: Nan Barber
Copyeditor: Gary Gorski
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

October 2019: First Edition

Revision History for the First Edition
2019-10-23: First Release

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Google. See our statement of editorial independence.

978-1-492-07602-5

[LSI]

Table of Contents

Preface

The Infrastructure Change Management (ICM) program at Google drives migrations, deprecations, and other large-scale infrastructure changes. Case studies in this report explore how infrastructure change projects are managed at Google. From these case studies, we'll provide insight into lessons learned from these different approaches, and provide an overview of techniques, processes, and tools that worked (or didn't work).

Acknowledgments

Sue Lueder, Brian Gustafson, Kate Ward, Salim Virji, Alex Matey, Derek Balling, Mike Koivunen, Patrick Bernier, Olivier Ansaldi, Shari Trumbo-McHenry, Janet Wu, Steven Schirripa, Simon O'Gorman, John Reese, and Sabrina Farmer.

Special thanks to Jessie Yang, and to the tech reviewers, Niall Richard Murphy and Grace Petegorsky, as well as to Betsy Beyer.

CHAPTER 1
Introduction

How do you as an engineering practitioner know if the change project you are managing qualifies as "infrastructure change"?

- Are the terms upgrade, migration, or decommission part of the change definition?
- Does your change affect multiple teams, organizations, products, and services within the company?
- Does your change impact engineering capabilities to maintain current plans, configurations, processes, or to apply software or policy changes?

If your answer to the above questions is yes, you are rolling out a large-scale infrastructure change.

We define *infrastructure change management* (ICM) as the execution of a planned, large-scale infrastructure change in order to increase project velocity, reduce cost, and lessen the overall pain inflicted on affected teams and customers.

A cliché, though apt, idiom for this kind of large-scale infrastructure change is "building the jet while flying it." Keeping the jet in flight and on course while building and rebuilding it requires an enormous amount of people to work as a team. If an engine dies, the crew needs to assess the situation, determine a corrective course of action, and ensure the safety of passengers onboard while communicating the issue in the right way, at the right frequency, to avoid widespread panic.

Large-scale infrastructure change works the same way, requiring coordination and communication with many teams, good processes and documentation, risk identification and management, monitoring, and tracking of the change progress. You can't ignore the low-probability but highly catastrophic events that can crop up mid-flight. Exercises like the Wheel-of-Misfortune[1] (disaster role playing) and DiRT[2] (annual event to push production systems to limit and inflict actual outages) are good ways to uncover these. The SRE Workbook also describes a number of organizational change management frameworks that may be useful to consider alongside infrastructure change.[3]

Infrastructure Change Management

These changes require strong processes and project management to ensure decisions are well-informed and communicated. The ICM program at Google, consisting of a dedicated team of technical program managers (TPMs), does just that: centrally driving migrations, deprecations, and other large-scale changes to infrastructure. Programs that ICM supports go through the following life cycle:

Concept Phase
Someone has an idea for a large-scale infrastructure change that could benefit from ICM support.

Backlog Phase
ICM performs a feasibility assessment of the concept proposal, compares its effort costs against the expected outcome, and ranks it in priority against other initiatives.

Planning Phase
People build an actionable project plan, publish target schedules, create objectives and key results for impacted teams, define key milestones and deliverables, and identify stakeholders and staffing. The goal of this phase is to take a concept proposal from the backlog and turn it into a work-about execution plan.

[1] For more info, see *https://landing.google.com/sre/sre-book/chapters/accelerating-sre-on-call/*.

[2] For more info, see *https://landing.google.com/sre/sre-book/chapters/lessons-learned/*.

[3] Consider using any of the frameworks referenced in the SRE workbook chapter: Organizational Change Management in SRE.

Execution Phase

In this phase, the project is under active execution. Impacted teams have product area (PA)–wide objectives and key results are centered around compliance with the program's goals.

ICM also provides dashboards to track infrastructure change progress across all active programs, as well as a tool called Assign-o-Matic that quickly maps production groups to best contacts. Many groups aren't associated with a product nor do they point to a human. Assign-o-Matic's heuristics for identifying the best contact solve the difficult and time-consuming problem of finding the right owner. Driving over a dozen active infrastructure change programs, ICM manages the complex network of dependencies that exist between them, so that the jet stays aloft with minimal-to-no impact to passengers.

One such program that ICM supported was the two-year MapReduce deprecation. MapReduce,[4] a flagship framework for large-scale data processing at Google, had been in maintenance mode since 2013. However, MapReduce usage continued to increase and by August 2017, users processed nearly 30 EiB of input and produced over 7 EiB of data. The goal of this infrastructure change program was to migrate all users off the MapReduce backend onto Flume, a higher-level application programming interface (API) built on top of MapReduce, which simplified expression of large-scale data computations.

Flume made it easier to build data-processing pipelines. The design goal was to make pipeline creation easier and more efficient. Rather than programming and tying together a series of independent MapReduce stages, we wrote one program with Flume and let it handle the execution details. By abstracting away from the low-level infrastructure, we did not need to work with all the underlying primitives —the panoply of data storage formats, parallel execution primitives, and job controller systems available at Google. Flume took care of all that, providing numerous benefits including reduced runtimes, less maintenance, and the ability for Google to focus on supporting a single platform for all users.

[4] More information is available online (https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf).

In August 2018, MapReduce was deprecated and replaced by Flume. During 2018, 50% of 30-day active build targets migrated off MapReduce and, by September 2019, over 45% of the remaining active targets were off MapReduce. As of 2019, Flume was rolled out to over 99% of C++ and Java pipelines, and the Flume support rotation was staffed with 12 engineers. Migrating to a new API and execution environment came at a cost to users. ICM helped minimize this cost and drive the migration alongside the many others in flight.

In this report, we provide two case studies on large infrastructure changes at Google: (1) a two-year effort to migrate all of the company's systems from Google File System (GFS) to Colossus and (2) a six-year effort to remove local disk storage for all jobs and move toward Diskless compute nodes. For each of these case studies, we provide an overview, the project's impact, the tools and processes used to manage the change, as well as individual lessons learned after each completed change. We conclude with a collection of key takeaways to consider when implementing a large-scale infrastructure change at your own organization. We hope that by sharing what worked and didn't work for us in these changes, other organizations may learn from our best practices and prepare for any anticipated risks that might occur along the way.

CHAPTER 2

Case Study 1: Moonshot

In this section, we discuss a large-scale project called Moonshot. We share several examples of tools, processes, and techniques that pushed the project forward and conclude with a postmortem of lessons learned from the project.

Overview

In 2010, the senior Storage SRE leadership declared that the Moonshot project would soon be underway. This project required teams to migrate all of the company's systems from GFS[1] to its successor, Colossus, by the end of 2011. At the time, Colossus was still in prototype, and this migration was the largest data migration in the history of Google. This mandate was so ambitious that people dubbed the project *Moonshot*. As an internal newsletter to engineers put it:

> If migrating all of our data in 2010 still sounds like a pretty aggressive schedule, well, yes it is! Will there be problems such as minor outages? Probably. However, the Storage teams and our senior VPs believe that it's worth the effort and occasional hiccup, and there are plenty of incentives for early adopters, including reduced quota costs, better performance, and lots of friendly SRE support.

The initial communication completely undersold the effort, complexity, and difficulty of this project. In reality, it took a full two years to migrate all of Google's services from GFS to Colossus.

[1] For an introduction to GFS, see this paper: *https://ai.google/research/pubs/pub51*.

GFS was designed in 2001 as Google's first cluster-level file system. It supported many petabytes of storage and allowed thousands of servers to interact with thousands of clients. Machines would run a daemon called a chunkserver to store blocks of data (chunks) on the local filesystem while the GFS client code split the files into a series of chunks and stored them on the chunkservers, replicating the chunks to other chunkservers for redundancy and bandwidth. The "GFS Master" kept the list of chunks for a given file along with other file metadata. GFS created shadow replicas so a new GFS Master could be selected if the primary one was down.

GFS limitations only started to surface roughly six years later. These were some of the limitations encountered:

- Google production clusters were larger, holding more than just thousands of machines.
- User-facing, "serving" systems like Gmail increasingly used GFS as the backend storage. Failures lasting minutes resulted in outages to these systems that were no longer acceptable.
- RAM stored the chunk locations and was limited by the maximum amount of memory you could physically put in a single machine.
- There was no persistent index of chunk locations, so restarting the GFS Master required recomputing the map of chunk locations. When a GFS cell (a virtual unit of measurement that represents a group of datacenter machines all managed by the same process) "restarted" for any maintenance reason, the master took 10–30 minutes to retrieve the full inventory of chunks.
- The GFS Master itself ran on a single machine as a single process. Only the primary master processed mutations of file system metadata. Shadow master processes running on other machines helped share the workload by offering read-only access to the GFS metadata.
- The GFS Master software couldn't take full advantage of SMP hardware because portions of the GFS Master software were single threaded. Plans to make software multithreaded were in the works but this option would still not be enough to meet the growing demand within a few years.

In early 2006, Google developers made available the initial implementation of Colossus—the eventual, though not originally intended, replacement for GFS. The developers had built Colossus with the specific purpose of being a backing store for giant BigTables.[2] By the summer of 2007, Colossus was developing into a proper cluster file system that could replace GFS as the cluster-level filesystem for BigTable. The developers set up the first production Colossus cell in January 2008, and videos began streaming directly from Colossus two months later.

Initially, many engineers were hesitant with the change to Colossus. Some resented the "mandate from above," feeling that they "had no choice." "This is a huge headache for us," another said in an interview.

Other engineers, however, were more optimistic. As Ben Treynor, VP of engineering, pointed out, "Moonshot is an extremely important initiative to Google. It is important enough that in order to make the fast progress necessary, we are willing to take greater risk in order to accomplish its goals—even to the point of taking outages, so long as we don't lose Customer data." This sentiment was summed up neatly by another engineer when they remarked, "It's crazy but it might just work."

Every year, Google's user base and services increased, and GFS could no longer support this ever-evolving ecosystem. We needed to move our systems to a cluster-level file system that was fault tolerant, designed for large-scale use, enabled more efficient use of machine resources, and provided minimum disruption to user-facing services.

To say that the migration was complex is an understatement. What started as a four-person team operation later turned into engineers volunteering their time as 20%ers[3] and, eventually, a dedicated 14–18 SRE team members per site to support the Colossus storage layer after the migration.

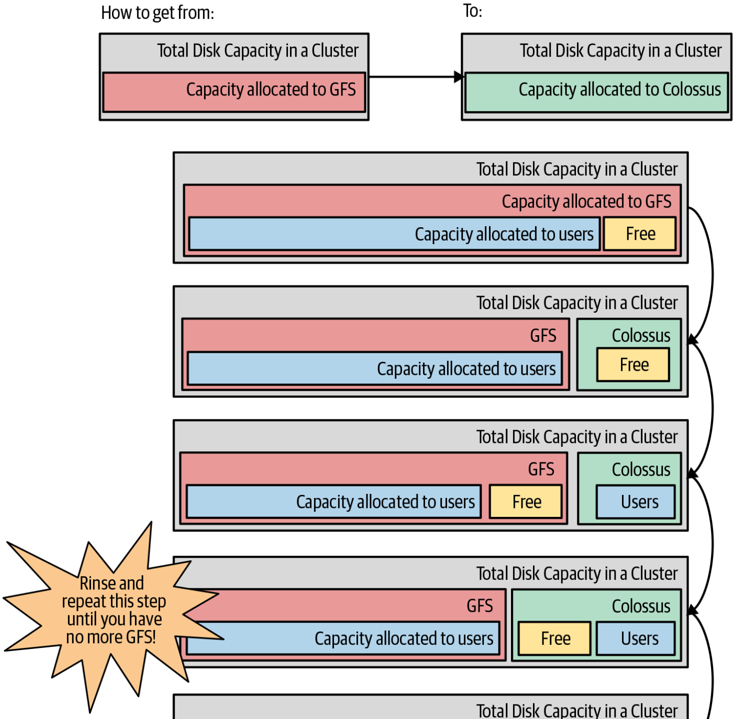
Pushing the project forward required most service owners to manually change their job configs to work with the new storage system. In other words, it required work from several SREs, software engi-

[2] For more info about BigTables, please read *https://ai.google/research/pubs/pub27898*.

[3] Dedicated 1 day (20%) of a 5-day week time for a personal project. See Chapter 5 in the book "The Google Way" by Bernard Girard: *http://shop.oreilly.com/product/9781593271848.do*

neers (SWEs), TPMs, SRE managers, product managers (PMs), and engineering directors. Moonshot also exposed issues with systemic resource usage spurring the "Steamroller" project, a split effort to reorganize how machine-level resources were allocated across the entire Google production fleet.

Below is a high-level, very simplified view of what the overall Moonshot migration entailed.


How to get from:
Total Disk Capacity in a Cluster
Capacity allocated to GFS
To:
Total Disk Capacity in a Cluster
Capacity allocated to Colossus
Total Disk Capacity in a Cluster
Capacity allocated to GFS
Capacity allocated to users
Free
Total Disk Capacity in a Cluster
GFS
Capacity allocated to users
Colossus
Free
Total Disk Capacity in a Cluster
GFS
Capacity allocated to users
Free
Colossus
Users
Rinse and repeat this step until you have no more GFS!
Total Disk Capacity in a Cluster
GFS
Capacity allocated to users
Colossus
Free
Users
Total Disk Capacity in a Cluster
Capacity allocated to Colossus
Capacity allocated to users
Free

Tools

As with any large-scale infrastructure change, creating the appropriate change structure and policies became critical to ensuring successful implementation of the change. Google SREs created migration tools that automated as much work as possible. These tools helped teams migrate successfully to Colossus with less effort, and some are described in detail below.

Quota and storage usage dashboard

A custom-built dashboard used to identify how much quota each team historically used. This was instrumental in showing the trending resource usage across machines, teams, and PAs. Note that "resource" here is defined as both storage and compute resources. For the Moonshot project, this identified where a GFS quota could be reclaimed for Colossus. This dashboard became so effective that it eventually turned into a widely accessed and supported tool for viewing resource utilization across the machine fleet.

Quota move service

A custom-built service designed specifically for Moonshot, to periodically free up quota from a source cell up to a minimum threshold, and add such freed quota to the destination cell.

This service enabled automatic, fine-grained moves of quota from GFS to D (discussed later in the Steamroller section) such that as the storage usage changed, the quota adjusted and kept appropriate headroom for the service maintained on both sides, to avoid disruptions. Think of the analogy of moving water from one bucket to another, except instead of just moving the water, the buckets were also resized (destination bucket grew bigger, source bucket grew smaller) so that each bucket had a similar amount of empty space in it at all times.

Migration planning and scheduling tool

A custom-built tool that provided a free quota loan when you moved, and created your scheduled migration window, intended at a time that was least disruptive to your service. This tool analyzed your file's directory structure, determined how to separate your data into chunks for moving, generated the namespace mapping files needed to copy data from GFS to Colossus, and generated the commands to perform the bulk data migration.

Migration tracking tool

A custom-built, frontend webserver, used to keep track of all GFS to D migrations for Moonshot, create or update migrations for users, and identify available capacity of each datacenter, for migration needs. The Moonshot team used this frequently for the execution and monitoring of the project throughout its phases and to communicate the data back to relevant stakeholders, for review.

Bulk data migration service

An internally built service used for bulk copying files from one location on GFS/Colossus to another destination in production. This is still used to this day for data copies of arbitrary sizes. The Moonshot team used it to move files off GFS to Colossus, one directory at a time.

These tools automated a non-trivial amount of work, and helped the team manage and track the migration, as well as the quota, so that team members would not have to do so manually themselves. The tools made the migration less troublesome and minimized human error. Besides tools, however, there were additional ways to make the migration easier. One way was to use processes to manage the migration.

Processes

We define processes as predefined methods of engagement, or methods used to execute the project. Moonshot used a number of processes within the team and outside the team to successfully move the project along.

In one internal process, the Moonshot team set up a weekly progress check-in meeting for those directly working on the project. People used this meeting to discuss updates on Colossus feature development, migration tooling development, Colossus rollout status, service migration status, risks and blockers on the critical path to be addressed, and so on. In some meetings, team members identified outdated documentation, missing procedures, and communication opportunities to increase awareness for affected customers. These turned into action items assigned to owners who followed up and provided an update at the next meeting if needed. In one instance, meeting attendees identified that the migration itself needed a separate meeting for more in-depth discussion, so they made that happen. Such a process facilitated communication within the team and helped team members manage the project more easily.

In another internal process, the Moonshot team divided the migration into phases, limiting the number of services it impacted at the same time. Video and Bigtable were the first customers on Colossus since GFS limitations hit them heavily and they were actively looking for a replacement storage system. Migrating these two early adopters helped the Moonshot team realize the time it would take to migrate a service at a per cell level and the tactical steps necessary for the migration (e.g., how to turn up a Colossus production and test cell, when to add and remove quota, etc.). Later on, they set up a pilot phase for two quarters, and a few smaller services (e.g. Analytics, Google Suggest, and Orkut) elected to migrate to Colossus. With each phase, the team discovered and addressed complexities before proceeding. Some lessons learned that were folded in later include defining what the common required migration tasks were per service, auditing file paths and permissions after migration was complete, understanding how much quota was needed to turn up a Colossus cell while the service was still running in GFS serving cell, and much more.

For processes external to the team, Moonshot created various communication channels to ensure ongoing feedback and prevent communication from getting lost in one person's inbox. Some examples of this included creating a dedicated project announcement mailing list and user list (e.g. moonshot-announce@google.com and moonshot-users@google.com), setting up office hours for one-on-one consulting, creating an exception procedure for folks who could not migrate by the targeted deadline, creating FAQs and migration instructions, and creating a form for users to submit feature requests or bug issues discovered. Each of these opened up transparency to questions and answers raised, and gave people a forum to collectively help one another. Office hours were set up to provide one-on-one consulting, for specific use cases.

Using these processes within the team and outside the team made information flow easier. These processes encouraged regular feedback, as well as communication and information sharing, within the immediate team and between teams. This kept the project moving and prioritized execution.

We've seen that processes are important for managing infrastructure change, but when the infrastructure change involves migrating huge amounts of data, we need to consider capacity as well. We talk about that in the next section.

Capacity Planning

Shortly after the senior Storage SRE leadership announced Project Moonshot, the Moonshot team discovered the storage migration from GFS and chunkserver-backed Colossus cells, to D-backed Colossus cells, required effectively creating CPU, memory, and storage resources out of thin air. The Borg[4] ecosystem had no visibility into GFS chunkserver resource accounting since GFS predated its time so there was not enough quota to turn up a D cell for the Moonshot migration. Therefore, the senior Storage SRE leadership announced the Steamroller project, an effort to address this problem, in an internal engineering announce list:

> "Because this is a prerequisite for the Moonshot migration, there is no "opt out" planned. In the case of extreme emergency, a limited deferment may be possible. Note: private cells and dedicated resources are exempted from this procedure, but are expected to have their own migration plans to accomplish Moonshot goals."

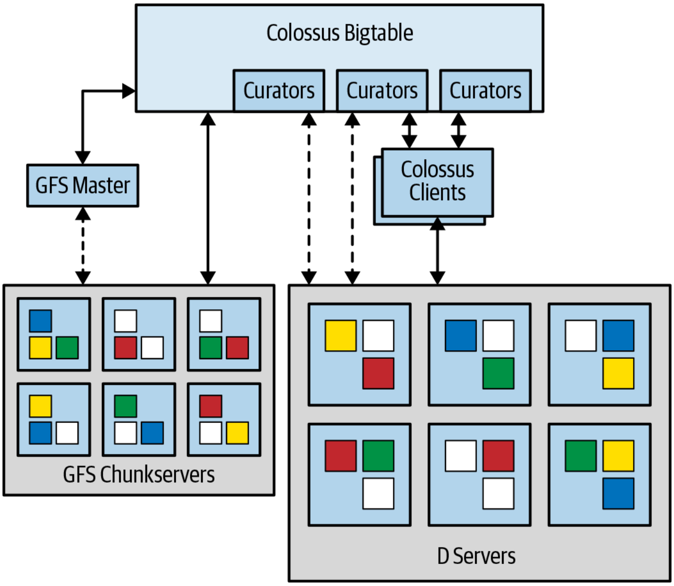
What is D?

Before we discuss the Steamroller project in more detail, we need to briefly introduce you to the D[5] (short for 'Disk') server, the equivalent of GFS Chunkservers, which runs on Borg. D formed the lowest storage layer and was designed to be the only application with direct read and write access to files stored on the physical disks. The physical disks were connected to the machines it ran on. Similar to how a GFS chunkserver managed reading and writing the raw chunk data while the GFS Master kept track of the filesystem metadata, the D server managed access to the raw data while Colossus managed the map of what data was on which D server. The Colossus client talked to Colossus to figure out which D servers to read/write from, and then talked directly to the D servers to perform the read/writes.

[4] Cluster management system for scheduling and managing applications across all Google data centers. For more info, see *https://ai.google/research/pubs/pub43438*

[5] For a high level storage overview, see *https://landing.google.com/sre/sre-book/chapters/production-environment/*

See below for a visual overview.



To get D-backed Colossus on Borg, there needed to be available storage resources to spin up D servers in each cell. The Moonshot team created a separate dedicated team across the development team and SREs, to acquire resources for D through the Steamroller project.

The Steamroller project

The Steamroller project primarily focused on recovering resources from the production fleet so that these resources could be repurposed to turn up D cells for Moonshot. The project covered rolling out general critical infrastructure changes as well, in order to minimize the number of disruptive events. This included rightsizing quota usage and applying Borg job resource limits, removing and preventing overallocation of cells, reinterpreting Borg job priorities on what gets scheduled first, and a few more.

In order to accomplish these goals, every team had to manually modify their configurations for all production jobs, one Borg cell at a time. They then had to restart their Borg jobs to match the new reallocation of capacity. The Borg job configurations had been written earlier when resources were plentiful and sometimes included generous padding for future growth possibilities. The modifications “steamrolled” these resources and also placed every Borg job into containers to impose these limits.

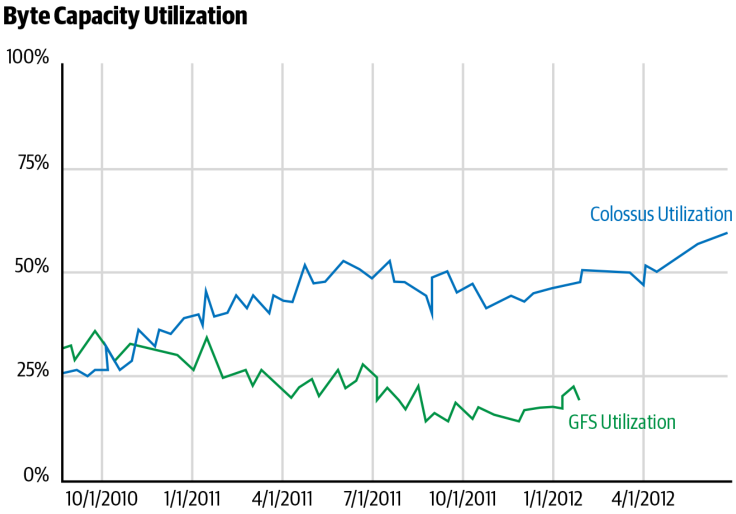
From a numbers perspective, the Steamroller project was successful. The team completed the project within the short span of a year, reclaiming a large amount of shared Borg resources from Borg jobs and returning a non-trivial amount back to the fleet. This enabled the Moonshot project to move forward since there were enough resources to allocate to D. From a change management perspective, however, several factors did not go well.

For example, the Steamroller team used a 90-day usage period to identify the 100th percentile for RAM and 99th percentile for CPU. The team used this as the new baseline measurement for each Borg job to be applied after restarting the jobs but did not take into account the small spikes in RAM usage. Therefore, if any tasks went over their new memory limits, even by a small amount, Borg killed them immediately, causing localized service disruptions and latency.

In another example, Google engineers felt that the Steamroller team communicated the project timeline in a pressing tone, which made engineers feel that they received notice of the upcoming change too late, and had limited information apart from the email. In hindsight, Moonshot’s initial schedule turned out to be overly optimistic—which caused the aggressive timeline—but this only became clear after a critical mass of work had already begun.

Finally, in one last example, the Steamroller project failed to get the initial staffing request of at least four full-time TPMs, and only received two TPMs. If the project had more TPMs, they could have engaged more with service owners during the initial service notification, exception review process, and the manual effort put in to overcome the lack of robust self-service tools. Management recognized the shortfall in staffing at the time and made decisions to delay communication of the project, limit automation effort, and reduce the scope of the project.

On a positive note, Steamroller did unblock the Moonshot project. The Moonshot project proceeded as shown in the Byte Capacity Utilization graph below. This graph shows the increase of Colossus utilization and the decrease of GFS utilization within the fleet.



Looking back on Moonshot and Steamroller, there were many lessons learned that may be useful for your own organization. We explore what we learned below.

Lessons Learned

The Moonshot project was the first large-scale data migration ever implemented at Google. Moonshot did not reach its advertised goal of moving off GFS to Colossus/D within one year but as Sabrina Farmer, VP of Engineering, mentioned, “[it] achieved several other very important but unstated goals. Focusing only on the advertised goal means we miss the other benefits a project brought.” Therefore, we hope these lessons learned shed insight into what went well and what did not, so you can apply them to your own infrastructure change.

The Moonshot project forced all teams to migrate by the target deadline.

> People felt they had no choice with this declared mandate. The team asked service owners to “swap out a known and tested storage system for one that was incomplete and had a comparatively low number of ‘road miles.’” Rather than forcing all teams to migrate by a target deadline, it would have been better to collaborate closely with the teams supporting complex services and allow smaller teams more time in the background to migrate.

Smaller teams often have less free capacity dedicated to taking on additional complex projects such as a migration.

The Moonshot team was comprised of 20%ers.

The migration team consisted of a handful of SREs, SWEs, PMs, and TPMs from various teams, who volunteered to work 20% of their time to make the migration happen. Such a distributed team meant they had both broad and deep levels of domain knowledge (for BigTable, GFS, D, and Colossus) to push the project forward. However, the fact that they were 20%ers and based in different offices in varying time zones added more complexity to the project. Eventually, management pulled in these SREs to work 100% on the migration effort and grouped them together, but it would have helped to have had a core migration team from the start. Nevertheless, if it's not possible to get what the team requested—such as what happened with Moonshot—you make do with what you have. Regarding Steamroller, one engineer summed this lesson up nicely when they said, "Develop for the team you have, not the one you're promised."

The Moonshot and Steamroller teams made conscious trade-offs between developing automation and meeting the deadline.

As a result of the aggressive deadline, the migration team did their best to automate the data migration as much as possible by using various tools, as we discussed earlier. What would have helped is using a robust workflow engine that automatically migrated data for the users, after fulfilling a set of requirements. Much of the migration required users to run commands, wait, and then run some more commands. Creating a workflow engine would have reduced the overhead for migrating. In the words of one TPM on the Steamroller project, "It was very hands on . . . we had to look at a lot of monitoring as the migration happened and then manually move." Before rolling out a large-scale infrastructure change in your organization, make a conscious assessment of the tradeoffs (in the context of the triple constraint model[6]) and understand what risks you will

[6] Time, scope, and cost information is available online (*https://www.pmi.org/learning/library/triple-constraint-erroneous-useless-value-8024*).

accept. This may expose any considerations around your project to an effective discussion.

Each change as part of the Moonshot project caused a rippling effect of customer frustration.

Even though the migration team had published information regarding the migration, Colossus's improved performance compared to GFS, and what people should expect from the migration, people still had questions and expressed frustration when the time came to migrate and when even more changes took place—such as the Steamroller project. Therefore, we learned that it's helpful to widely advertise a large-scale infrastructure change through many different channels, for example, large, Google-wide, technical talks; office hours; user mailing lists; and company-wide announcements. Keep in mind, however, that there will always be someone unhappy with the change. The best you can do is reduce the blast radius of the change to affected users. You do this by automating as much of the manual work as possible, getting support from your technical influencers (i.e., your tech leads, engineering managers, or others of hierarchical seniority) to help disseminate information and by utilizing the different communication channels mentioned.

We applied these lessons learned from Moonshot to several infrastructure change management projects we worked on afterward, including our next case study, Diskless.

CHAPTER 3

Case Study 2: Diskless

From 2012 through 2018, Technical Infrastructure (TI) teams rolled out a Google-wide change to production: to remove local disk storage for all jobs and move toward Diskless compute nodes, aka cloud disks.[1] Such resource disaggregation reduced cost through improved server platform availability, tail latency, and disk utilization. This move to "prod without disk," the vision of Technical Leads Eric Brewer, Luiz Barroso, and Sean Quinlan, was principally motivated by the following considerations:

- The performance of a spinning disk was growing at a slower rate than that of CPU, SSD, or networking. Over time, the amount of storage "trapped" behind an interface increased faster than the speed of that interface.
- Network-attached storage enabled migration of compute across machines, without losing storage while hugely accelerating the physical maintenance of machines. Storage-specific hardware eliminated the barrier to network-attached storage.
- Separating compute and storage devices improved tail latency. Previously, hundreds of jobs competed for limited disk quota and bandwidth at the same time. In a Diskless world, you

[1] Luiz A. Barroso, Urs Hölzle, and Parthasarathy Ranganathan, "Disk Trays and Diskless Servers." Section 3.4.1 in *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. 3rd ed. (Morgan&Claypool, 2019), 66.*https://books.google.com/books?id=b951DwAAQBAJ*

scheduled I/O with quotas and did parallel reads; that is, you sent three parallel read requests and used the first one that came back and canceled the others ("best of three").

- Independently provisioning compute and storage on different cycles improved datacenter total cost of ownership (TCO) and the ability to scale.
- On shared machines, spinning disk reliability decreased compute reliability; 25%–30% of production task deaths were attributable to disk failure.
- Removing one resource type (local disk) simplified the provisioning and configuration of services that had to be managed.

This section talks about the Diskless effort and what tools and processes we used for this project. We conclude with what worked and didn't work for Diskless, as well as lessons learned.

Overview

Diskless was a multi-year, multi-project effort to enable Google's software stack to run on platform servers without local hard drives. As these diskless servers began to percolate throughout the fleet, the software stack had to run in a Diskless environment. Otherwise, jobs would not schedule and run, and services would become unavailable.

Google had historically used two types of production servers: index and diskfull. Index servers had one or two Serial AT Attachment (SATA) disks, and diskfulls had six or more. Storage-specific hardware, deployed in 2013 and considered Google's first storage appliance, delivered disk servers that provided remote storage without local compute. Before this, each disk in a tray ran Borg to coordinate resources and tasks running on the local machine. Now with the storage-specific hardware, Borg and compute tasks did not run on local spinning disks, but elsewhere in the cell. This configuration optimized the storage service and hardware, providing significant tail latency improvements, and opened the door to resource disaggregation.

Though production servers always had at least one hard drive, teams generally avoided using it in the serving path. Specifically, the service needed to continue serving requests despite an all-too-common local disk failure. This consideration was reinforced with the move to Borg, Google's cluster management system. Borg did not offer durable storage for tasks. Instead, that function was provided by Colossus, Bigtable, and other storage systems. Most Borg local disk use was limited to a handful of standard cases: binary staging, logging, and miscellaneous "scratch" usage. This made it easy—theoretically—to excise disk from the picture and move to Diskless compute nodes.

By 2014, Platforms Engineering[2] approved the diskless server platform and Google accelerated software development work to support it. Two years later, job conversion to Diskless began rolling out, reaching two-thirds of production, and diskless server machines started arriving in Borg.

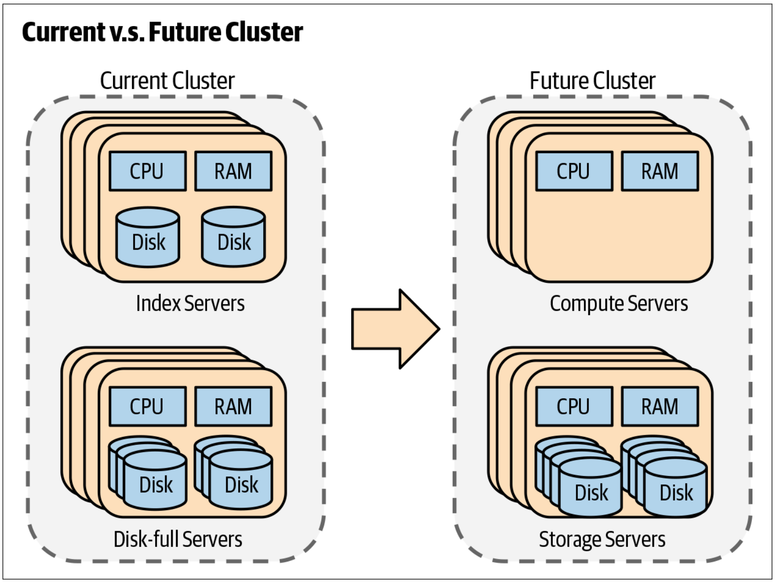


Figure 3-1. Diskless architecture change overview

The Diskless migration plan revolved around the aforementioned use cases of local disk. At the heart of Diskless was the base software stack—13 software sub-tracks that effectively replaced the local disk with a remote option, or some flavor of a remote procedure call (RPC) system. The primary goals of these sub-tracks were to:

[2] The team that designs, develops, deploys, tests, and supports hardware and software for Google's data centers and delivers Google's global computer.

- Support the transition to Diskless compute nodes.
- Minimize new dependencies for serving jobs.
- Keep per-team migration costs to a minimum.

Actuating these tracks and moving to a Diskless-ready world required some additional tooling, described below.

Jet Part Swapping

The overall migration process provided two paths to Diskless: (1) an explicit conversion for sophisticated teams that preferred direct control and (2) an "autoconversion" option for all others. Cell conversion essentially ensured that when jobs requested a legacy disk, they were given a RAM filesystem instead. Autoconversion meant that the system applied this "bait and switch" administratively, unbeknownst to the user. Autoconversion was enabled cell by cell, prior to the arrival of Diskless machines in the cell, and was how most jobs became diskless ready.

Additional control was added, by allowing the automatic option to be turned on or off manually. The Diskless team never touched jobs directly; automatic changes altered defaults and took effect the next time a user touched a job.

The migration team used a tool called Subspace to notify users about actions they needed to take, to make their jobs Diskless ready. Subspace was the centralized notification and routing system for Google's infrastructure. This tool made it easy to send notifications for thousands of jobs, across every product at Google, about production resources—without maintaining a database of contacts.

In terms of project management, the team used Google Sheets to track the majority of the project artifacts: per-product area notes, contacts, status, timelines and schedules for the base software track, migration-blocking bugs, lists of blocked users, and week-over-week Diskless adoption statistics. Dashboards supplanted many of these and made it easier to track base rollout status and diskless adoption across Google (broken down by job, CPU usage, owner, and product area, providing a much-needed, fine-grained targeting of users).

Once built, the tooling and processes worked well. Unfortunately, tooling, docs, and dashboards were built mostly in 2016, only after the program ran into trouble. By then, the timeline was getting too protracted, and core engineering teams were burning out. The primary four members of the Diskless team had been fielding one migration question after another. As one team member put it:

> "[over the next year] we were getting multiple user support questions a day for migration (sometimes dozens). The fan-in situation wasn't sustainable. Because everyone waited until a few specific quarters to migrate, [the rush arrived] and when that rush comes, you can't just spin up team members like VMs."

What Didn't Work?

Most production changes to support Diskless were non-events or straightforward for most teams to implement. One aspect worth mentioning, though, was that the migration itself (building the core software and core functions) was not the largest burden. In the words of one lead engineer on the project:

> "It was [as if] . . . the larger company had built up cross product of hundreds of frameworks, config languages, resource planning systems, quota management systems, automation frameworks, scripts, custom monitoring, etc. And you implicitly "own" the problem of every integration point a third party team has built (in the expectations of the users)."

In addition to this complexity, the effort stumbled on many fronts, including staffing, planning, communication, and risk management.

Staffing

Similar to Moonshot, except for a few sub-projects, the Diskless project was insufficiently staffed. People who could have influenced staffing changes did not realize what was necessary for the company to provide staffing. Perhaps this could have been addressed by better training on how to ask for more staffing on a project. For example, this affected the debug logging side of Diskless. One engineer put it this way:

> "We had to do volunteer rotations across our wider logs team to consult with people and explain to them how to move in to the new version because the support load of the migration overwhelmed the team."

Regardless, for an effort of this size and complexity, senior program managers should have been added from the beginning. In 2016, additional VPs joined the effort and enlisted TPMs across every PA to bolster it. By the end of 2016, over two-thirds of all Borg jobs were Diskless ready.

Planning

Due to the huge complexity and scope of this project, Diskless required more effort and staff than anticipated. This led to schedule slips, difficulty getting teams involved, and lack of understanding on when and how to act. Moreover, although the diskless platform arrived on a set schedule, the software implementation schedule shifted frequently. Software systems needed to be Diskless ready before their users could migrate. Before a jet can take off, all parts—passenger compartment, wings, cockpit, all infrastructure—need to be ready too.

The Diskless project introduced support for remote debug logs (RDLs) primarily to decouple the lifespan of logs from the lifespan of individual jobs and have the logs be independent of whether a task or alloc stayed active. Initially, the system garbage collected the logs as soon as a job was done, so people missed the opportunity to view the logs if they were not timed well. The engineering effort and ongoing cost of disk usage for remote debug logs versus the cost of alternative hardware—such as SSD—wasn't deeply considered or accounted for during initial design decisions. RDLs and the teams that operated them added significant cost in terms of time and money. In addition, the migration was complex for power users with large, revenue-critical applications at Google, further exacerbating the situation, and put those users and revenue at risk.

Moreover, the core Diskless team did not thoroughly collect requirements on some critical user journeys (such as debug logging), which resulted in a solution with different features and behaviors than those of the local disk. Some of these—easy task searching and filtering, new debugging, and analysis capabilities that were not previously possible—were wins but others led to regressions and new challenges for SREs operating and debugging the services.

Communication

Subspace worked well for communicating necessary production changes, and the Diskless team used the tool to make numerous public announcements of upcoming changes. In spite of this, however, the Diskless effort as a whole lacked a clear communication plan. Due to the volatile schedule, the migration team found it difficult to communicate firm dates, along with any urgency for requested work. It was only later in the project, when additional TPM support was brought in, that there was a clear engagement with the PAs and impacted teams. At this point, dashboards and reporting also became available, and they provided significant visibility on the migration progress.

Unexpected challenges

"Most teams had to twiddle a bunch of things," was the comment from one engineer. Becoming Diskless ready required a lot of time and work, particularly for those who had local disk usage outside the norm. It was difficult to track down these teams well, nor align on priorities with them and their management chain. Moreover, many teams who used disk in "normal" ways ran into small issues. Many things lurk under the hood of a complex code base with a normal level of technical debt.

Lessons Learned

The Diskless effort was not an easy project by any means. Here are some of the lessons we learned along the way.

Infrastructure change protocol

Updating and evolving basic infrastructure was hard and clarity was needed on guidelines, priorities, and engagement protocols so that the most revenue-critical, risk-averse clients weren't the guinea pigs.

Clear communication

We should have communicated clearly to the impacted teams and the entire management chain. The motivation for the change should be clear to those doing the work; teams should be able to explain why they're doing it. VPs and directors needed to be briefed on the program, timelines, priorities, and expected range of impacts to teams. As in the case with Moonshot, support from technical influencers was a key factor. When things were broken—staffing included—we should have escalated. It was critical that the right TPM staff engaged early on to ensure continuous and effective communication.

Avoid hard deadlines

The design path chosen for Diskless made deadlines hard and inflexible, and the consequences of missed deadlines severe—jobs would not schedule and Google would go down. If the software wasn't ready when machines showed up, it was a bad thing. At the time, Google's processes were not prepared to couple hardware and software so tightly. There were hard deadlines *and* we needed to mobilize the entire company. SREs, whose mission it is to keep the system reliable, bore the brunt of that pressure. The dynamic range of constraints meant that any change deadline you chose was a bad one. It would have been better to use a phased approach, gaining experience with the change as you go.

Managing a major change across every team at Google when a hard deadline was approaching was impossible. To put this into perspective, Google's culture empowered every engineer, regardless of their position, to have local incentives and a strong sense of shared values in engineering. This made managing a project from the top down (unless it was universally urgent) extremely difficult. People resisted supporting the change because top-down project management was not Google culture. As one engineer put it:

> "We really didn't see much real momentum until people realized the deadline was deadly serious . . . although this aggressive deadline was controversial among affected stakeholders, subjectively it did seem to help create a sense of urgency and 'skin in the game' that was not present before."

Looking back, one of the reasons Google has become more successful is precisely because it managed to decouple the software and hardware from one another despite the number of setbacks. Diskless was one such large program that helped accomplish this. In the next section, we'll dive into what common lessons you can take away from both studies as well as others we've identified that can be practical tips to consider.

CHAPTER 4
Preflight Checklist

The lessons learned here are specific to each of the case studies examined, but may be modified to fit your individual needs. We've also listed 10 general guidelines to keep in mind when implementing a large-scale infrastructure change at your organization. Regardless of the size of your organization, applying these key takeaways may help mitigate the challenges of rolling out a large infrastructure change.

1. **Establish a core team (if it doesn't yet exist) to manage the infrastructure change in the company**

   Staff the effort with the right people from the beginning to ensure projects smoothly launch and land. At a minimum, there should be full-time engineers (to build code for the migration tools and assist with answering questions), technical project managers (who facilitate communication, tracking, and meeting deadlines), and an executive sponsor (who helps push this change at the top, to ensure it gets prioritized).

2. **Pilot with the more technically savvy, low-risk customers first**

   These customers are more aware of what features they need and can provide useful feedback to improve the migration before a large rollout. In addition, try to select the customers that are considered to be low risk (i.e., unexpected issues would not stop operations for them).

3. **Understand trade-offs up front as much as possible**

While establishing a core team is critical, it's not always possible to have dedicated people working on the migration project or, perhaps, the program complexity was not well understood at the outset. Clarify at the start the lost opportunity costs in the project, such as delivery delays, low quality project communication, or unmanaged migration risks for critical services. By doing such clarification up front, the team can identify and proactively accept the risks brought in due to these constraints.

4. **Understand your customer requirements**

   Before the change project starts, gather user requirements to see what specifically they want in a system and for what purpose. Even if their use cases will not be built in the same way in the new system, it helps to ensure you're building the right tools for the right audience, to ensure a smooth migration. As you gather the requirements, you may also come across unique corner cases. By gathering corner-case situations up front, you frontload your risk and ensure you have sufficient slack in the schedule, to either prioritize the requirements or collaborate with the team to adjust their workflows so that it works with the new system.

5. **Publish your plan of record**

   A plan of record confirms the project plan and key decisions, as agreed on by the project stakeholders. This includes, at a minimum, a glossary of key vocabulary, project goals, project timeline, and key milestones, with assigned owners. It's essential to have one source of truth to revisit, when plans change. Within Google, we share this plan of record broadly, both inside and outside the project team. In doing so, this provides transparency in what teams can expect from the project and transparency for how decisions were made.

6. **Push the migration out in phases**

   The migration itself is a disruption to service operations. Even with a plan in place, significant risks still exist. Staging the migration in phases relevant to the scale of your organization is an effective way of implementing the change. As issues emerge during earlier phases, you have time to update tools, techniques, and processes before the risks impact more services in later phases. An example of a phased migration approach could be early/

alpha testing, voluntary migration, assisted migration, forced migration, and then deprecation of the old service.

7. **Automate as much of the manual, repeatable process as possible**

   Depending on how large the infrastructure change is and how many people it affects, automating relevant processes saves time for engineers, so they can focus on more complex issues, and avoids burdening users with manual and toilsome work. For example, in Moonshot's case, a migration scheduling tool was built to identify an appropriate time for the migration to take place. This tool took into account cluster maintenance time and launch dates for the service. Think about how much time it would take for you to build an automated tool to perform a process, and how much time you would spend manually performing the process for each service. This helps you determine the return on investment (ROI) of creating a tool versus manually handling the process.

8. **Test early and often**

   Having a testing environment setup for users, to test whether their service functions on the new infrastructure, is critical for uncovering and mitigating technical risks. Testing should simulate, as closely as possible, the behavior that the production environment offers, when services are migrated. Any deviation from that behavior exposes more risk.

9. **Communicate early and often**

   For a large-scale infrastructure change, issues may crop up at any time. Those leading the implementation of the change must continually communicate early and often about this change and through the right channels. People are often frustrated when any change occurs, and more so if it was not communicated clearly enough or to the right people. Therefore, communicate early and often to reduce the resistance to the change. Some examples of how you can do this include creating an FAQ, sending announcements to relevant internal engineering newsletters or mailing lists, presenting at company-wide all-hands meetings, offering one-on-one consulting, and creating a landing project page containing relevant information.

10. **Create appropriate escalation and exception procedures**

It’s not uncommon for a service to need an exception or extension to a large-scale infrastructure project. This occurs because a change may not have the features a team needs, because there are conflicting and committed project deadlines, or for other valid reasons. Regardless of the underlying context, providing escalation and exception procedures ensures that teams are aware of the proper channels, to communicate and collaborate with the change team. When creating these procedures, gather details such as the name of the service requesting the extension, how much more time they would need, and the justification for such an extension.

CHAPTER 5
Conclusion

The case studies we've discussed provide two tales of foundational, large-scale infrastructure change: one of replacing Google's distributed file system with its successor; the other of decoupling storage and compute. In both instances, the jet (or, at least, an engine) was rebuilt mid-flight and, in both cases, at least one lesson was clear: think about all the ways your infrastructure change can break your users, then don't let the infrastructure change break them.

This requires a pre-flight checklist and some paranoia: know how things can go wrong, consider the whole spectrum of bugs, from benign to catastrophic, and their range in rarities.

Regardless of the type of infrastructural jet you're piloting and fixing along the way, we hope these case studies will be as instructive for you as they were for us.

About the Authors

Wendy Look has been at Google since 2012, starting out with the internal Information Technology support team (aka Techstop) and is now a Technical Program Manager in Site Reliability Engineering (SRE) at Google Switzerland. She holds a BA degree in Japanese and Chinese from Hamilton College, and is a current MS candidate at Bentley University studying Human Factors in Information Design. She currently lives in Zurich, Switzerland, spending every sunny weekend enjoying the mountains.

Mark Adam Dallman is a Technical Program Manager in Site Reliability Engineering (SRE) at Google, specializing in infrastructure change and capacity management. He's studied mathematics, philosophy, and computer science at the University of Wisconsin-Madison and Columbia University. In his spare time, he enjoys cycling and playing speed chess.