Tracking and Controlling Microservice Dependencies

DEPENDENCY MANAGEMENT IS A CRUCIAL PART OF SYSTEM AND SOFTWARE DESIGN.

SILVIA ESPARRACHIARI, TANYA REILLY, AND ASHLEIGH RENTZ

In search of a cappuccino, cheese bread, and a place to check her email, Silvia walked into a coffee shop. Upon connecting to the Wi-Fi hotspot, a captive portal prompted her to log in and offered a few third-party authentication options. When she clicked on one of the access token providers, her browser showed a "No Internet Connection" error. Since she didn't have access to the network, she couldn't get an OAuth token—and she couldn't access the network without one.

This short story illustrates a critical detail of system design that can easily go unnoticed until an outage takes place: cyclic dependencies.

Dependency cycles will be familiar to you if you have ever locked your keys inside your house or car. You can't

open the lock without the key, but you can't get the key without opening the lock. Some cycles are obvious, but more complex dependency cycles can be challenging to find before they lead to outages. Strategies for tracking and controlling dependencies are necessary for maintaining reliable systems.

REASONS TO MANAGE DEPENDENCIES

A lockout, as in the story of the cyclic coffee shop, is just one way that dependency management has critical implications for reliability. You can't reason about the behavior of any system, or guarantee its performance characteristics, without knowing what other systems it depends on. Without knowing how services are interlinked, you can't understand the effects of extra latency in one part of the system, or how outages will propagate. How else does dependency management affect reliability?

SLO

No service can be more reliable than its critical dependencies.[8] If dependencies are not managed, a service with a strict SLO[1] (service-level objective) might depend on a back end that is considered best-effort. This might go unnoticed if the back end has coincidentally high availability or low latency. When that back end starts performing exactly to its SLO, however, it will degrade the availability of services that rely on it.

High-fidelity testing

Distributed systems should be tested in environments that replicate the production environment as closely as

possible.[7] If noncritical dependencies are omitted in the test environment, the tests cannot identify problems that arise from their interaction with the system. This can cause regressions when the code runs in production.

Data integrity

Poorly configured production servers may accidentally depend on their development or QA (quality assurance) environments. The reverse may also be true: a poorly configured QA server may accidentally leak fake data into the production environment. Experiments might inadvertently send requests to production servers and degrade production data. Dependency management can expose these problems before they become outages.

Disaster recovery / isolated bootstrap

After a disaster, it may be necessary to start up all of a company's infrastructure without having anything already running. Cyclic dependencies can make this impossible: a front-end service may depend on a back end, but the back-end service could have been modified over time to depend on the front end. As systems grow more complex over time, the risk of this happening increases. Isolated bootstrap environments can also provide a robust QA environment.

Security

In networks with a perimeter-security model, access to one system may imply unfettered access to others.[9] If an attacker compromises one system, the other systems that depend on it may also be at risk. Understanding how systems are interconnected is crucial for detecting and

limiting the scope of damage. You may also think about dependencies when deploying DoS (denial of service) protection: one system that is resilient to extra load may send requests downstream to others that are less prepared.

DEPENDENCY CYCLES

Dependency cycles are most dangerous when they involve the mechanisms used to access and modify a service. The operator knows what steps to take to repair the broken service, but it's impossible to take those steps *without* the service. These control cycles commonly arise in accessing remote systems. An error that disables `sshd` or networking on a remote server may prevent connecting to it and repairing it. This can be seen on a wider scale when the broken device is responsible for routing packets: the whole network might be offline as a result of the error, but the network outage makes it impossible to connect to the device and repair it. The network device depends on the very network it provides.

Dependency cycles can also disrupt recovery from two simultaneous outages. As in the isolated bootstrap scenario, two systems that have evolved to depend upon each other cannot be restarted while neither is available. A job-scheduling system may depend on writing to a data-storage system, but that data-storage system may depend on the job-scheduling system to assign resources to it.

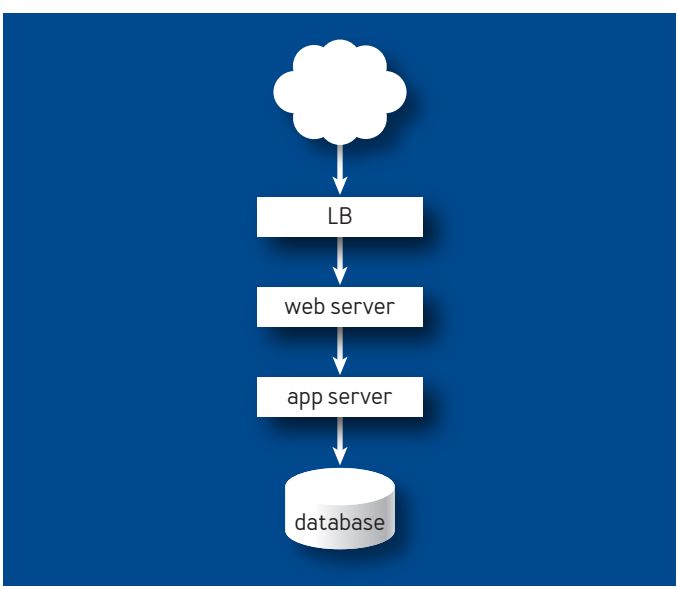
Cycles may even affect human processes, such as oncall and debugging. In one example, a source-control system outage left both the source-code repository and documentation server unavailable. The only way to get to

the documentation or source code of the source-control system was to recover the same system. Without this key information about the system's internals, the oncall engineer's response was significantly obstructed.

MICROSERVICES AND EXTERNAL SERVICES

In the era of monolithic software development, dependency management was relatively clear-cut. While a monolithic binary may perform many functions, it generally provides a single failure domain containing all of the binary's functionality. Keeping track of a small number of large binaries and storage systems is not difficult, so an owner of a monolithic architecture can easily draw a dependency diagram, perhaps like that in figure 1.

FIGURE 1: **SAMPLE DEPENDENCY DIAGRAM**



The software industry's move toward the microservices model makes dependency management much more difficult. As Leslie Lamport said in 1987, "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."[5] Large binaries are now frequently broken into many smaller services, each one serving a single purpose and capable of failing independently. A retail application might have one service for rendering the storefront, another for thumbnails, and more for currency conversion, checkout, address normalization, surveys, etc. The dependencies between them cross failure domains.

In her 2017 Velocity NY Conference talk, Sarah Wells of the *Financial Times* explained how her development teams manage more than 150 microservices—and that is for just one part of the *Financial Times's* technical estate. Squarespace is in the process of breaking down its monolith[4] and already has more than 30 microservices. Larger companies such as Google, Netflix, and Twitter often have thousands of microservices, pushing the problem of dependency management beyond human capabilities.

Microservices offer many advantages. They allow independent component releases, smoother rollbacks, and polyglot development, as well as allowing teams to specialize in one area of the codebase. They are not easy to keep track of, however. In a company with more than a hundred microservices, it is unlikely that employees could draw a diagram and get it right, or guarantee that they're making dependency decisions that won't result in a cycle.

Both monolithic services and microservices can

experience bootstrapping issues caused by hidden dependencies. They rely on access to decryption keys, network, and power. They may also depend on external systems such as DNS (Domain Name System). If individual endpoints of a monolith are reached via DNS, the process of keeping those DNS records up to date may create a cycle.

The adoption of SaaS (software as a service) creates new dependencies whose implementation details are hidden. These dependencies are subject to the latency, SLO, testing, and security concerns mentioned previously. Failure to track external dependencies may also introduce bootstrapping risks. As SaaS becomes more popular and as more companies outsource infrastructure and functionality, cyclic dependencies may start to cross companies. For example, if two storage companies were to use each other's systems to store boot images, a disaster that affected both companies would make recovery difficult or impossible.

DIRECTED ACYCLIC GRAPHS

At its essence, a service dependency is the need for a piece of data that is remote to the service. It could be a configuration file stored in a file system, or a row for user data in a database, or a computation performed by the back end. The way this remote data is accessed by the service may vary. For the sake of simplicity, let's assume all remote data or computation is provided by a serving back end via RPCs (remote procedure calls).

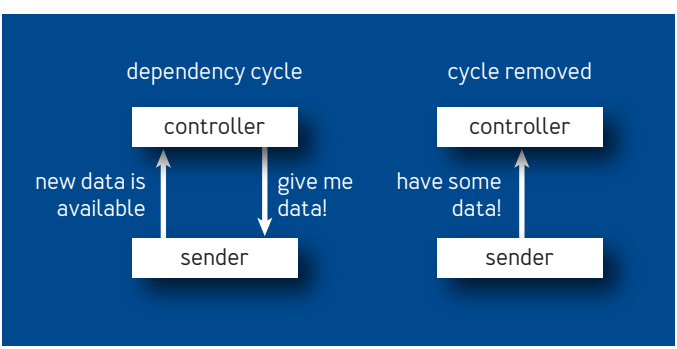
As just seen, dependency cycles among systems can make it virtually impossible to recover after an outage.

The outage of a critical dependency propagates to its dependents, so the natural place to begin restoring the flow of data is the top of the dependency chain. With a dependency cycle, however, there is no clear place to begin recovery efforts since every system is dependent on another in the chain.

One way to identify cycles is to build a dependency graph representing all services in the system and all RPCs exchanged among them. Begin building the graph by putting each service on a node of the graph and drawing directed edges to represent the outgoing RPCs. Once all services are placed in the graph, the existing dependency cycles can be identified using common algorithms such as finding a topological sorting via a depth-first search. If no cycles are found, that means the services' dependencies can be represented by a DAG (directed acyclic graph).

What happens when a cycle is found? Sometimes, it's possible to remove a cycle by inverting the dependency, as shown in figure 2. One example is a notification system where the senders notify the controllers about new data,

FIGURE 2: **CYCLE REMOVAL**



and the controller then pulls data from the senders. The cycle here can be easily removed by allowing the senders only to push data into the controller. Cycle removal could also be accomplished by splitting the functionality across two nodes—for example, by moving the new data notification to a third system.

Some dependencies are intrinsically cyclic and may not be removed. Replicated services may periodically query their replicas in order to reinforce data synchronization and integrity.[3] Since all replicas represent a single service, this would be represented as a self-dependency cycle in the graph. It's usually okay to allow self-dependencies as long as they do not prevent the isolated bootstrapping of the system and can properly recover from a global outage.

Another intrinsically cyclic dependency occurs in data-processing pipelines implemented as a workers-controller system.[2] Workers keep the controller informed about their status, and the controller assigns tasks to the workers when they become idle. This cyclic dependency between workers and controllers may not be removed without completely changing the processing model. What can be done in this case is to group workers and controllers into a supernode representing a single service. By repeating this edge contraction for all strongly connected components of the graph, taking into account their purpose and practical viability, you may achieve a DAG representation of the original graph.

TRACKING VS. CONTROLLING

In some environments, you can derive great benefit from just understanding the existing dependency graph. In

others, determining the existing state is not sufficient; mechanisms are needed for preventing new undesirable dependencies. The two approaches examined here, dependency tracking and dependency control, have different characteristics:

➡ *Tracking dependencies is a passive approach.* You use logging and monitoring to record which services contact each other, then look back at that data in the future. You can understand the dependencies by creating data structures that can be queried efficiently or by representing the relationships visually.

➡ *Controlling dependencies is an active approach.* There are several points during design and implementation where you can identify and avoid an undesirable dependency. Additionally, you can prevent connections from being made while the code is running in production. If you wait until the dependency has already been used and monitored, it will be too late to prevent the issues it may cause.

These approaches overlap (e.g., data collected during dependency control can certainly be used for tracking), but let's look at them separately.

Dependency tracking

Initially, dependency tracking often takes the form of information stored in engineers' heads and visualized in whiteboard drawings. This is sufficient for smaller environments, but as the system becomes more complex, the map of services becomes too complicated for any one person to memorize. Engineers may be surprised by an outage caused by an unexpected dependency, or they may not be able to reason about how to move a service

and its associated back ends from one data center to another. At this stage, organizations begin to consider programmatically generated views of the system.

Different environments may use different ways of collecting information about how services are interconnected. In some, a firewall or network device might record logs of which services are contacting each other, and these logs can be mined for dependency data. Alternatively, a set of services built on a common framework might export standard monitoring metrics about every connection; or distributed tracing might be used to expose the paths a request takes through the system, highlighting the connections.

You can aggregate whatever sources of information are available to you and create a dependency graph, processing the data into a common structure and optimizing it for running queries over it. From there, you can use algorithms on the graph to check whether it is a DAG, visualize it using software such as Graphviz and Vizceral, or expose information for each service, perhaps using a standard dashboard with a page for each service.

By continually monitoring traffic between systems and immediately integrating it into the graph, new dependencies may be seen shortly after they reach production. Even so, the information is available only after the new dependency has been created and is already in use. This is sufficient for dependency *tracking*, where you want to describe the interconnections of an existing system and become aware of new ones. Preventing the dependency, however, requires dependency *control*.

Dependency control

Just like dependency tracking, dependency control typically starts as a manual process using information stored in engineers' heads. Developers might include a list of proposed back ends in all design documentation and depend on their colleagues' knowledge of the existing systems to flag dangers. Again, this may be enough for a smaller environment. As services are born, grow, change, and are deprecated, the data can quickly become stale or unwieldy. Dependency control is most effective if enforced programmatically, and there are several points to consider in adding it.

When working on dependency management at Google, we found it best to think about controlling dependencies from the client side of a client-server connection (i.e., the service that is about to depend on another service). By owning the code that initiates the connections, the owner of the client has the most control and visibility over which dependencies exist and can therefore detect potential problems earlier. The client is also most affected by ill-considered dependencies.

Although the owner of a server may want to control who its clients are for reasons such as capacity planning or security, bad dependencies are much more likely to affect the client's SLO. Because the client requires some functionality or data from the server for its own functionality or performance, it needs to be prepared for server-side outages. The server, on the other hand, is unlikely to notice an outage of one of its clients.

One approach to dependency control is to analyze the client's code and restrict dependencies at build time.

The behavior of the binary, however, will be influenced by the configuration and environment it receives. Identical binaries might have very different dependencies in different situations, and the existence of code inside a binary is not a reasonable predictor for whether that binary has a dependency on another service. If a standard mechanism is used for specifying back ends or connection types—for example, if all back ends are provided in configuration and not in code—this might be an area worth exploring.

Restrictions are most effective if applied at runtime. By intercepting and potentially blocking connections as they are being made, you can be certain that you are inspecting the actual behavior of the running system, rather than speculating based on code or configuration. To avoid wasted engineering effort, restrictions on the back ends that services may contact should be implemented as early in the development life cycle as possible. Changing a system's architecture after it's already live and in use is far more expensive.[6] By applying the same set of restrictions at all stages of software development—during development, testing, canarying, and running live—any unwelcome dependency can be identified early.

There are several options for runtime enforcement. Just as with dependency tracking, existing infrastructure could be repurposed for dependency control. If all interservice connections pass through a firewall, network device, load balancer, or service mesh, those infrastructure services could be instrumented to maintain a list of acceptable dependencies and drop or deny any requests that don't match the list. Silently dropping requests at a point between the client and server may complicate debugging,

though. A request that is dropped for being an unapproved dependency may be indistinguishable from a failure of the server or the intermediate device: the connections may seem to just disappear.

Another option is to use a dedicated external dependency-control service that the client can query before allowing each new back-end connection. This kind of external system has the disadvantage of adding latency since it requires extra requests to allow or deny each back end. And, of course, the dependency-control service itself becomes a dependency of the service.

At Google, we had the most success adding restrictions into the code at the point where the connection is made. Since Google had a homogenous environment with a standard RPC mechanism used for all connections, we were able to modify the RPC code to match each new back end against a "dependency control policy"—an extra configuration option provided to every binary.

AUTHORIZING RPCS

The dependency-control policy consists of an ACL (access control list) of the RPC names expected to be initiated by a service. For performance reasons, the policy is serialized and loaded by the service during startup. If the policy is invalid (because of syntax errors or data corruption), it's not used and the dependency control isn't activated. If the policy is correct, it becomes active, and all outgoing RPCs are matched against it. If an RPC is fired but isn't present in the policy, it will be flagged as rejected. Rejected RPCs are reported via monitoring so that service owners can audit them and decide on the correct course of action: remove

the RPC from the binary if it's not a desired dependency or add it to the ACL if it's indeed a necessary new dependency.

This pseudocode shows how the authorization of RPCs could be implemented:

```
func isAllowedByPolicy(rpc, acl):
  foreach expectedRPC in acl:
    if rpc == expectedRPC:
      # If the RPC is listed in the ACL,
      # it should be allowed.
      return true

  # If the RPC didn't match any item on the ACL,
  # it should be rejected.
  return false
```

To prevent production outages, service owners are allowed to choose whether to enforce the policy and drop rejected RPCs or to soft-apply the policy and allow rejected RPCs to go through. Most service owners choose to enforce the policy in their test and QA environments so they catch new dependencies before they reach production, then soft-apply the policy in production. Even when soft-applying the policies, monitoring and alerting are still available for RPCs that would be rejected.

As mentioned before, the ACL can be based on historical information about RPCs fired by the binary, but that implies allowing the binary to run and serve production data for some time without dependency controls. Also, depending on the variability and diversity of the outgoing traffic, some RPCs might be rare or fired only under special circumstances, such as turn-down or crash. In this

case, they might not show up in the historical data and would have to be manually added to the ACL. Because of this, service owners would be well advised to run in soft-apply mode for a long time before enforcing dependency controls in production.

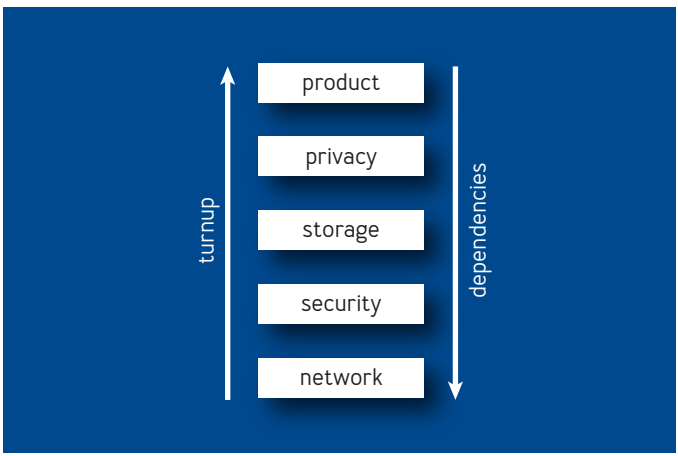
ISOLATING GROUPS OF SERVERS

Authorizing RPCs by name is a good way to control RPCs that are uniquely served by a single back end, but this will not cover all of the dependency issues highlighted earlier. One example is a data-integrity case where the same RPC is served by both production and test servers. Unless the policy can distinguish between the serving back ends, you cannot block production RPCs from reaching test instances. Additionally, ACLs can offer a tight lock around dependencies, but they aren't singly sufficient to prevent dependency cycles.

To prevent these other dependency issues, RPCs can be isolated within a group of servers. The first decision to be made is choosing a DAG model that will dictate the communication between the sets of servers. One simple model that prevents cycles is a graph that represents the expected turn-up sequence of the servers. This is a *stacked* or *layered* model of the system, shown in figure 3. The layered model reinforces that servers will never have dependencies on layers higher than the ones they live on, enabling the sequential bootstrap of the servers from bottom to top. Servers can depend on servers only at the same layer or below.

Services at the bottom layer can rely only on local static data in order to bootstrap, never on data served by

FIGURE 3: **STACKED MODEL**



another service. At Google, the full set of static data that's necessary to bootstrap a system (e.g., compiled binaries, configuration files, system documentation, root keys, etc.) is called the *backpack*.

A layer can have sublayers in order to prevent dependency cycles between servers in the same layer. Sublayering is a smaller problem and can often be handled by a single team without coordination with other teams.

The layered model works well for enabling the bootstrap of a single system, but it still doesn't solve the problem of isolating production and test environments, or restricting communications among different geographic regions. To tackle this kind of problem, you must be able to group servers into disconnected sets; this is called the *isolated model*. Unlike the layered model, where dependencies are allowed in the downward direction, the isolated model disallows dependencies among different

FIGURE 4: **ISOLATED MODEL**

components. In the example in figure 4, the products Jupiter, Earth, and Mars are not allowed to exchange RPCs with each other. Thus, they are not allowed to depend on each other.

One way to generalize dependency authorization in a DAG model is to let oriented edges represent *can-send-to* relations. Each node on the graph has a self-referencing edge (i.e., they can send RPCs to themselves). Also, the can-send-to relation is transitive: if A can send RPCs to B, and B can send RPCs to C, then A can send RPCs to C. Note that if B can send RPCs to A, and B can send RPCs to C, that does not imply that A can send RPCs to C or vice versa. Can-send-to is a directed relation. If there were a can-send-to relation in both directions (from A to B and from B to A), this would constitute a cycle and the model wouldn't be a DAG.

The pseudocode for authorizing RPCs in a DAG model could be written as:

```
func isAllowedByModel(rpc, model):
  clientNode = model.resolveNode(rpc.sender)
  serviceNode = model.resolveNode(rpc.receiver)
  return model.hasTransitiveConnection(clientNode, serviceNode)
```

The isolated model can be combined with the layered model, allowing the isolated bootstrap of each region to be reinforced, as illustrated in figure 5.

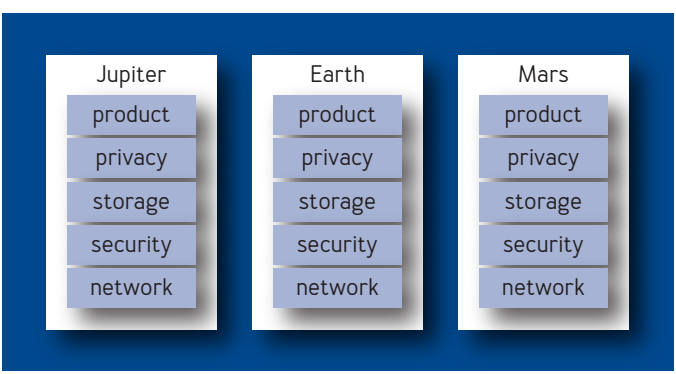
The pseudocode for combining different models can be as follows:

```
func isAllowedByAllModels:
  foreach model in modelCollection:
    # Checks if the RPC is allowed by the model.
    if !isAllowedByModel(rpc, model):
      return false

  # If no models reject the RPC, then it should be allowed.
  return true
```

Be careful when combining models that you don't isolate critical components by combining mutually exclusive models. Usually, simple models are easier to understand and to predict the results of combining, like the layered and isolated models described here. It can be

FIGURE 5: **COMBINED MODEL**



challenging to predict the combined logic for two or more complex models. For example, suppose there are two models based on the geographical locality of machines. It's straightforward to see that assigning locality "Tokyo" from one model and locality "London" from the other model will result in an empty set, since no machine can be physically located in London and Tokyo at the same time. Meanwhile, if there are two tree models based on locality—such as one for city, time zone, and country, and another for metro, voting zone, and country—it might be difficult to verify which combinations of values will return non-empty sets.

Related articles

- The Hidden Dividends of Microservices
  Microservices aren't for every company, and the journey isn't easy.
  Tom Killalea
  https://queue.acm.org/detail.cfm?id=2956643
- A Conversation with Werner Vogels
  Learning from the Amazon technology platform
  https://queue.acm.org/detail.cfm?id=1142065
- Fail at Scale
  Reliability in the face of rapid change
  Ben Maurer, Facebook
  https://queue.acm.org/detail.cfm?id=2839461

CONCLUSION

With the growth of massive interdependent software systems, dependency management is a crucial part of system and software design. Most organizations will benefit from tracking existing dependencies to help model their latency, SLOs, and security threats. Many will also find it useful to limit the growth of new dependencies for data integrity and to reduce the risk of outages. Modeling infrastructure as a DAG will make it easier to be certain there are no dependencies that will prevent isolated bootstrapping of a system.

Dependencies can be tracked by observing the behavior

of a system, but preventing dependency problems before they reach production requires a more active strategy. Implementing dependency control ensures that each new dependency can be added to a DAG before it enters use. This gives system designers the freedom to add new dependencies where they are valuable, while eliminating much of the risk that comes from the uncontrolled growth of dependencies.

References

1. Beyer, B., Jones, C., Petoff, J., Murphy, N. R. (Eds.). 2016. *Site Reliability Engineering: How Google Runs Production Systems*, 37-40. O'Reilly Media.
2. Beyer, B., Jones, C., Petoff, J., Murphy, N. R. (Eds.). 2016. *Site Reliability Engineering: How Google Runs Production Systems,* Chapter 25: "Data Processing Pipelines." O'Reilly Media.
3. Chang, F., et al. 2006. Bigtable: a distributed storage system for structured data; https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf.
4. Kachouh, R. 2017. The pillars of Squarespace services. Squarespace Engineering; https://engineering.squarespace.com/blog/2017/the-pillars-of-squarespace-services.
5. Lamport, L. 1987. Email message sent to a DEC SRC bulletin board; https://www.microsoft.com/en-us/research/publication/distribution/.
6. Saini, A. 2017. How much do bugs cost to fix during each phase of the SDLC? Synopsis; https://www.synopsys.com/blogs/software-security/cost-to-fix-bugs-during-each-sdlc-phase/.

7. Seaton, N. 2015. Why fidelity of environments throughout your testing process is important. Electric Cloud; http://electric-cloud.com/blog/2015/09/why-fidelity-of-environments-throughout-your-testing-process-is-important/.
8. Treynor, B., Dahlin, M., Rau, V., Beyer, B. 2017. The calculus of service availability. *acmqueue* 15(2); https://queue.acm.org/detail.cfm?id=3096459.
9. Ward, R., Beyer, B. 2014. BeyondCorp: a new approach to enterprise security. *;login:* 39(6), 6-11; https://ai.google/research/pubs/pub43231.

Silvia Esparrachiari Ghirotti *has a bachelor's degree in molecular science and a master's in computer vision and human-computer interaction from the University of São Paulo. She started at Google as a software engineer 8 years ago and has worked in social products, user data privacy and fighting abuse. She currently leads the team developing tools for dependency control.*

Tanya Reilly *is the principal engineer for infrastructure at Squarespace. She previously spent 12 years improving the resilience of low-level services at Google, including introducing a layered model for dependency control. She speaks at conferences about site reliability and software failure, and blogs at noidea.dog.*

Ashleigh Rentz *is a technical writer whose interests include blameless post mortems and wearable technology. She spent 14 years at Google in various roles, most recently producing internal documentation for SRE and Google Cloud Platform.*