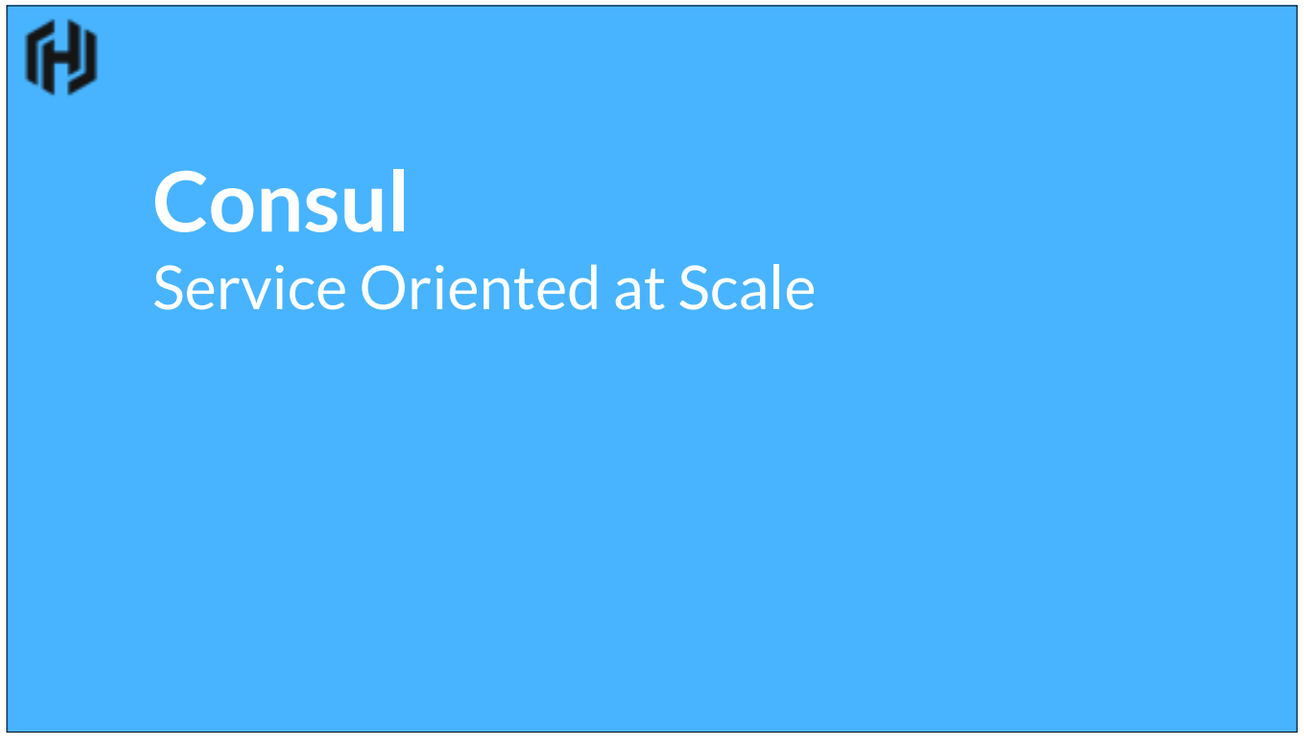

Consul
Service Oriented at Scale

Armon Dadgar

@armon

My name is Armon Dadgar, and you’ll find me around the internet as just @armon

I am a co-founder of HashiCorp, which focuses on building tooling for a software-managed datacenter.

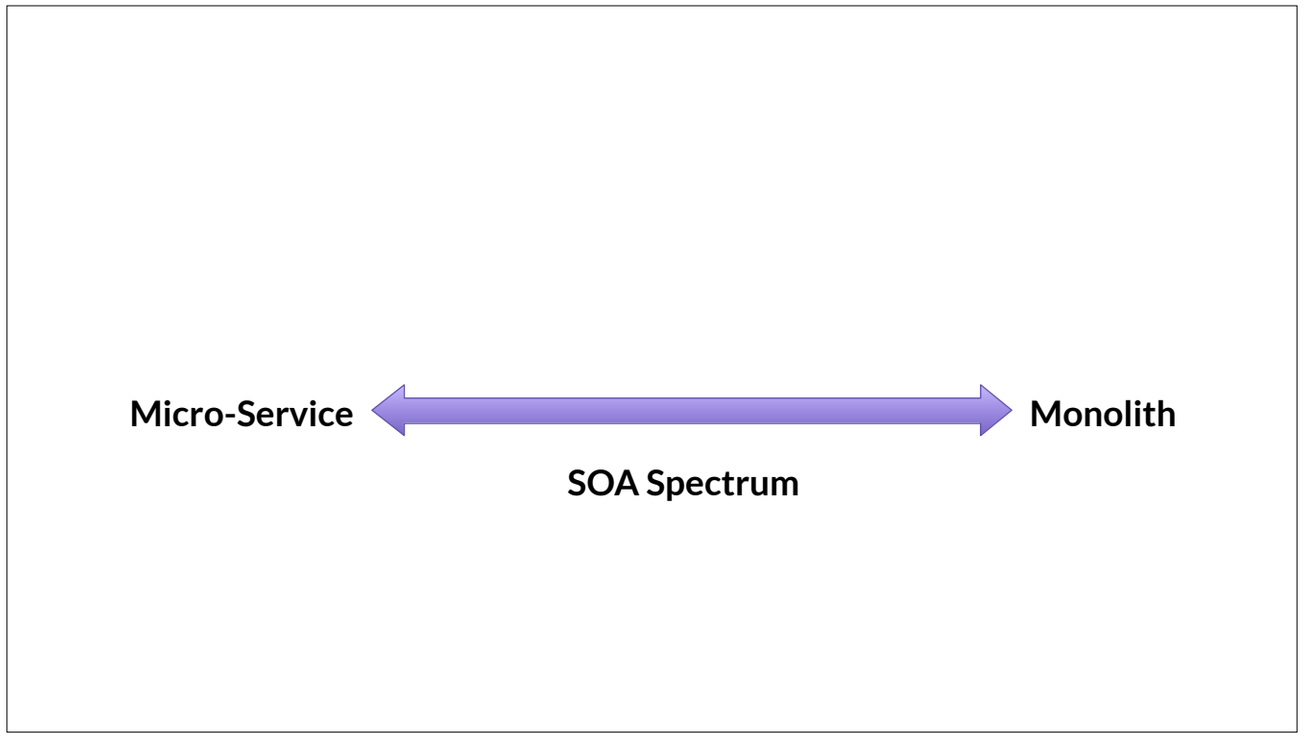
You may recognize some of our open source tools such as Vagrant, Packer, Serf, Consul and Terraform.

Service Oriented At Scale

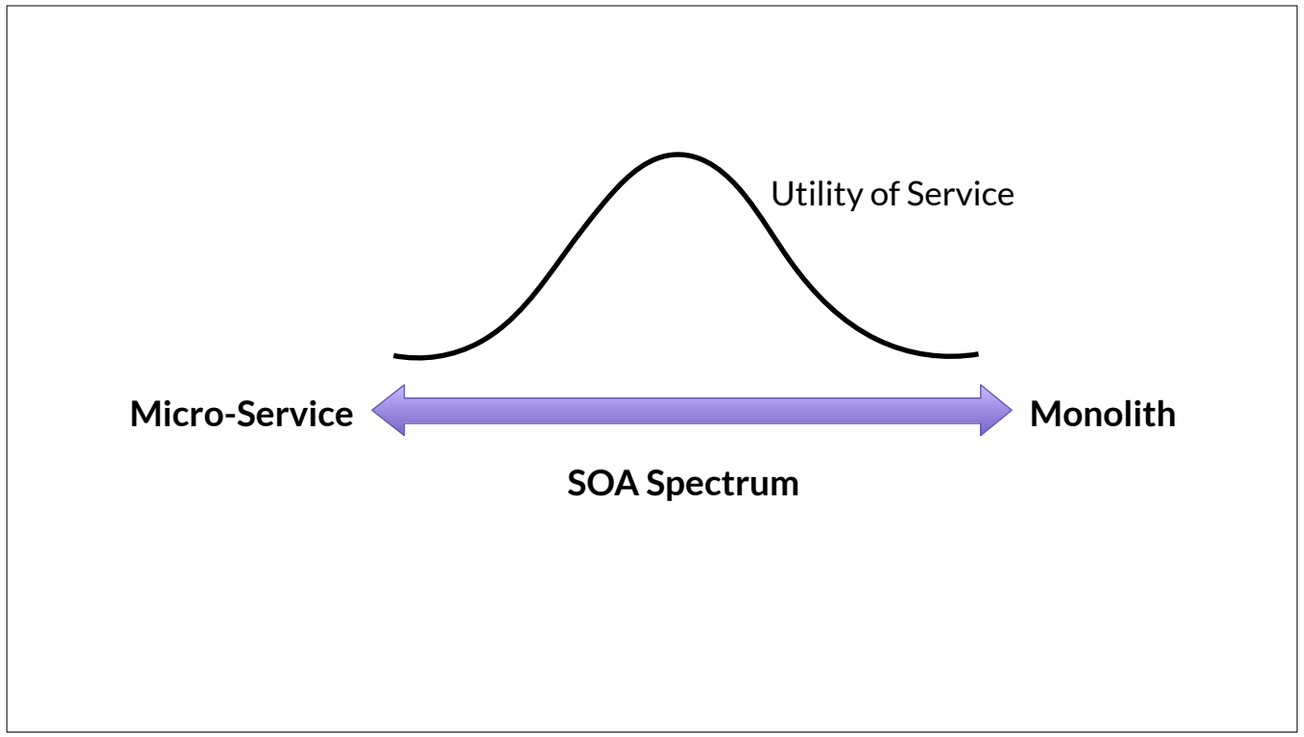
Alphabet Soup

- Service Oriented Architecture (SOA)
- Micro-Services
- Monoliths
- N-Tier Architectures

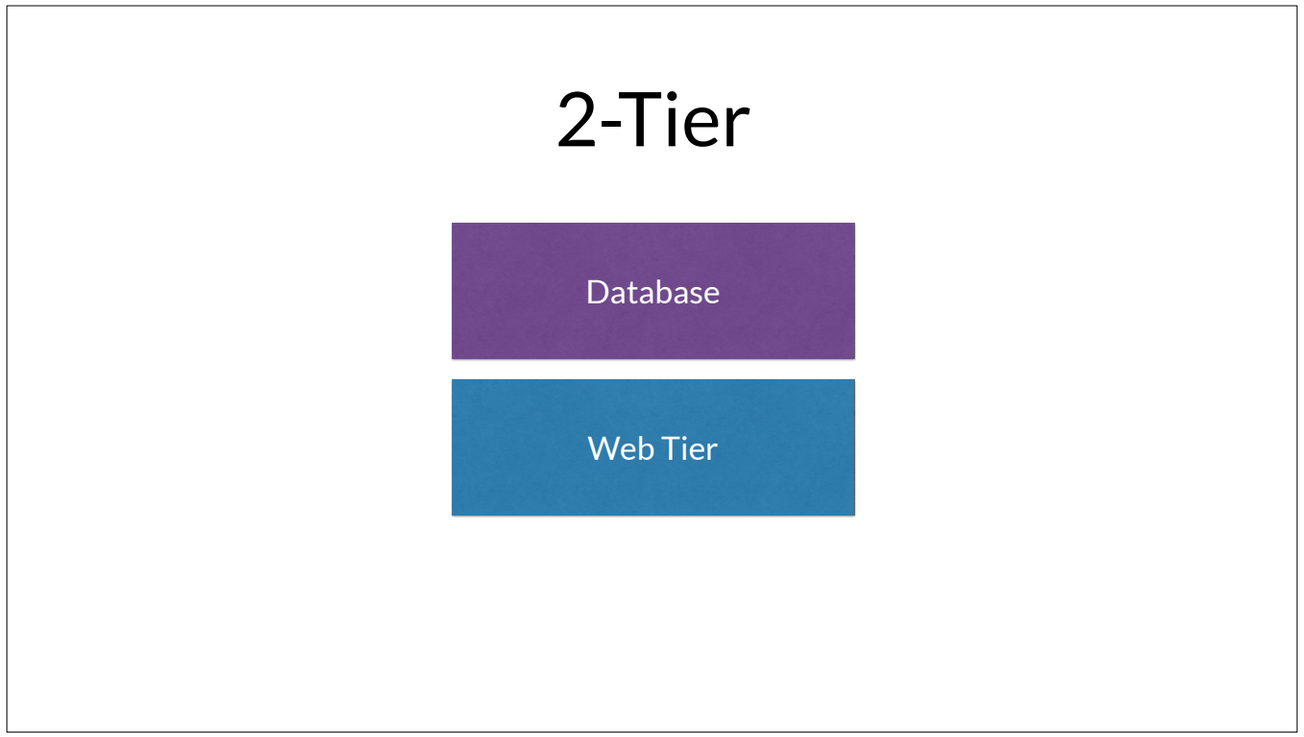
Today we are talking about Service Oriented at Scale. Unfortunately, we now have a buzzword soup when discussing anything in this space. You may have heard of terms like Service Oriented Architectures, SOA, Micro-Services, Monoliths, and N-Tier architectures.



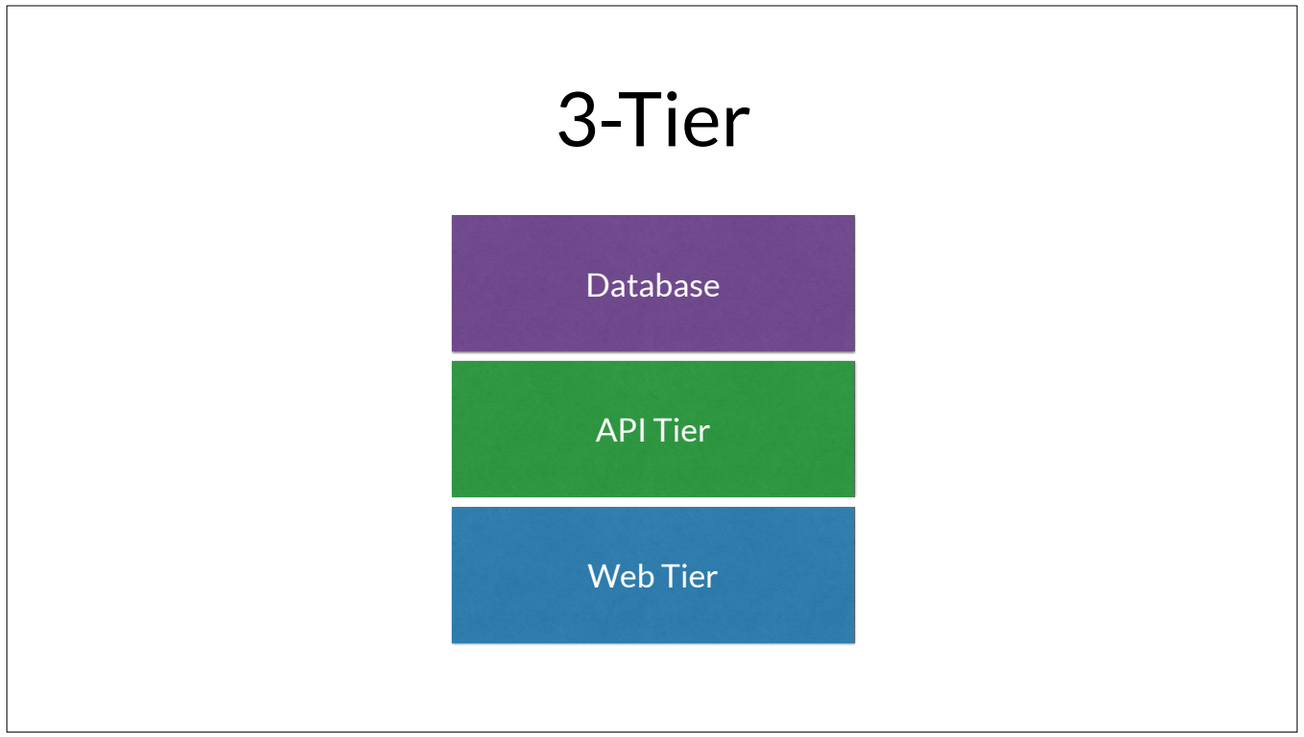
As with most things, SOA is a spectrum. On one side is “Micro-Services”. These are the smallest unit, some define it as 100 LoC, or 1 RPC method. On the other side we have the “Monolith”. This involves packaging the entire application into a single enormous service.



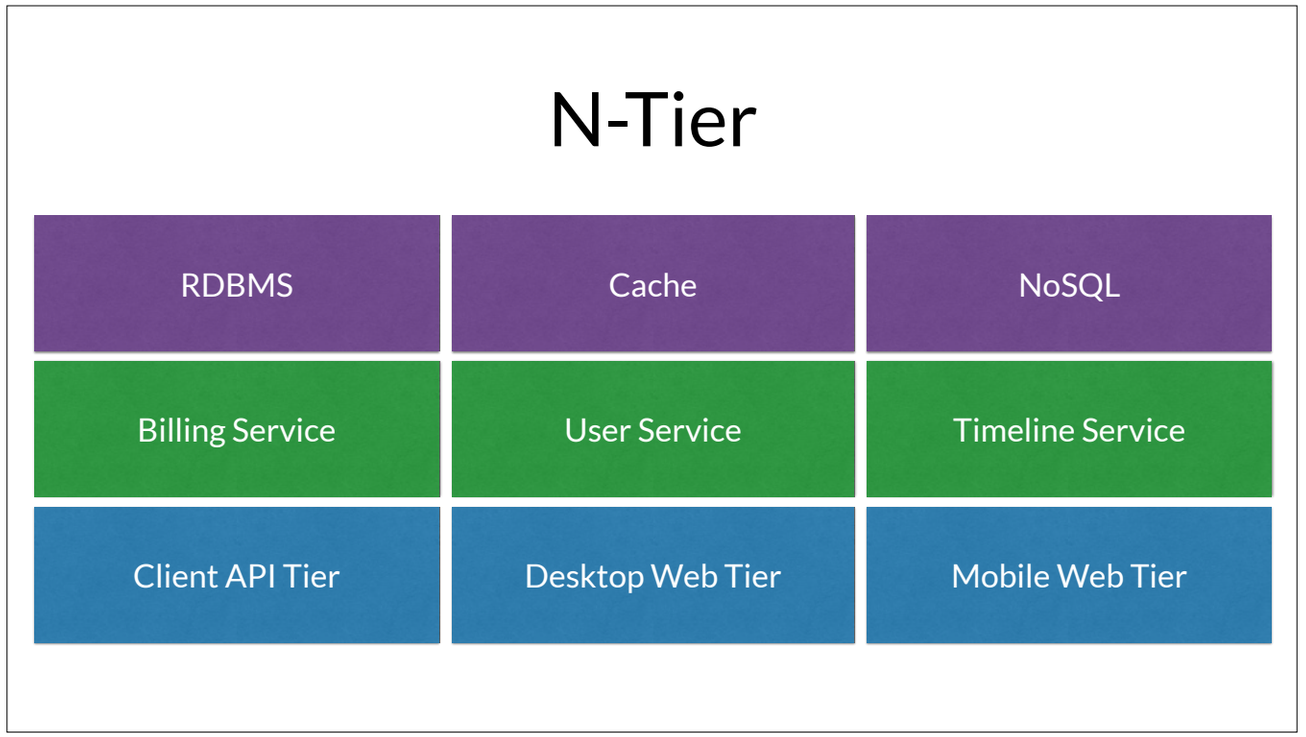
I’d argue that the utility curve of a service, is probably normally distributed along the spectrum. Services that are too small introduce both conceptual overhead of needing to understand a web of hundreds or thousands of services, while having a few enormous applications makes it very hard to reason about the workings of that particular service.



In terms of the N-Tier architectures, I think examples are most useful. The most basic example would be a simple 2 tier application. Here we may have a Web and Data tier.



This might eventually evolve into the classic 3 tier architecture. Here we introduce an intermediate API tier which separates presentation from business logic.



Eventually, the final evolution of this is the N-Tier architecture. Here there are multiple presentation tiers, multiple services containing different bits of business logic and many data tiers that are optimized for their particular access patterns and semantics.

SOA Goals

- Autonomous
- Well-Defined (Limited) Scope
- Loose Coupling

More fundamental however is the goals of SOA. Again many definitions are possible, but the one I prefer is that a service should be as autonomous as possible, it's scope should be well-defined but limited. This means you can't define the scope of your service as "Twitter". Lastly, there should be a loose coupling between services.

SOA Benefits

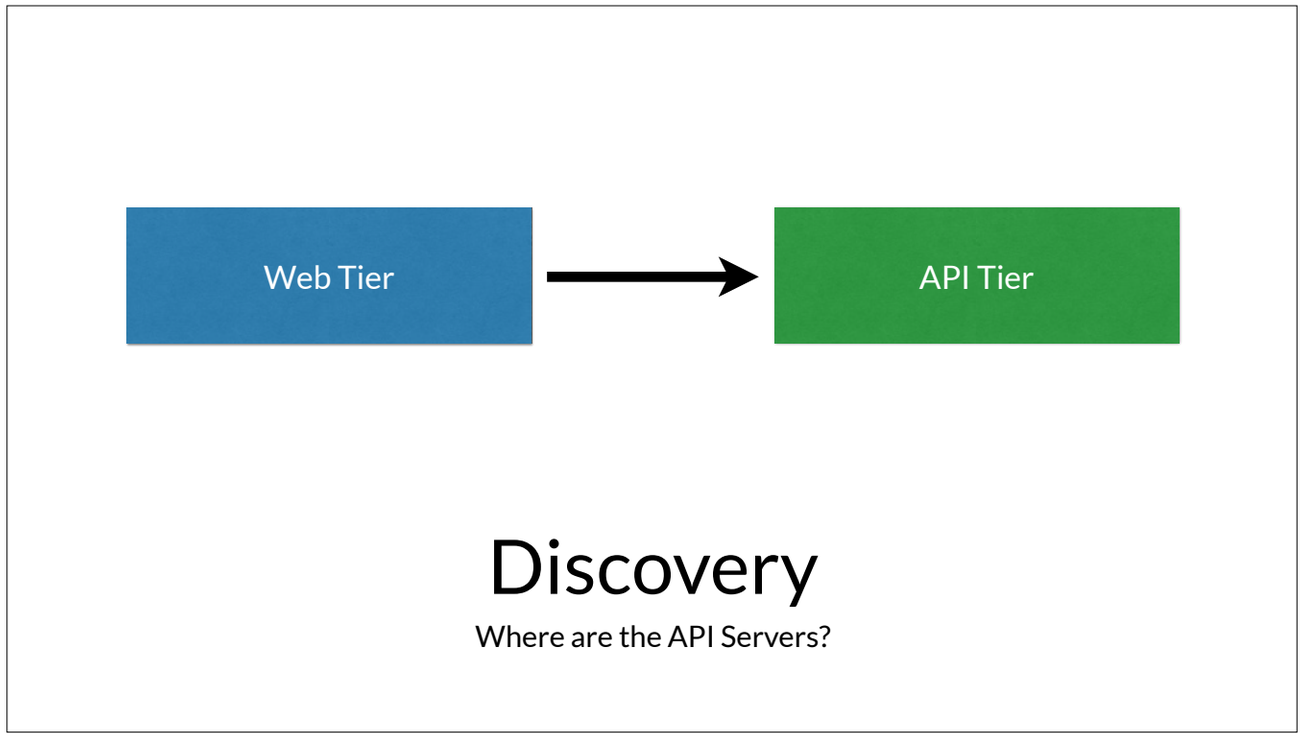
- Develop Faster
- Composable
- Failure Isolation
- Testable
- Scale Out vs Scale Up

Given that we stick to our goals, using a SOA architecture has many benefits. In many ways this is analogous to writing a single enormous function or decomposing into many sub-routines. We can develop faster since we can reason about the problem domain. Services can be composed together to build higher level applications. We can isolate failure domains to specific services. Each service becomes more testable as we know the inputs and outputs. Lastly, services enable a scale out instead of a scale up approach as well.

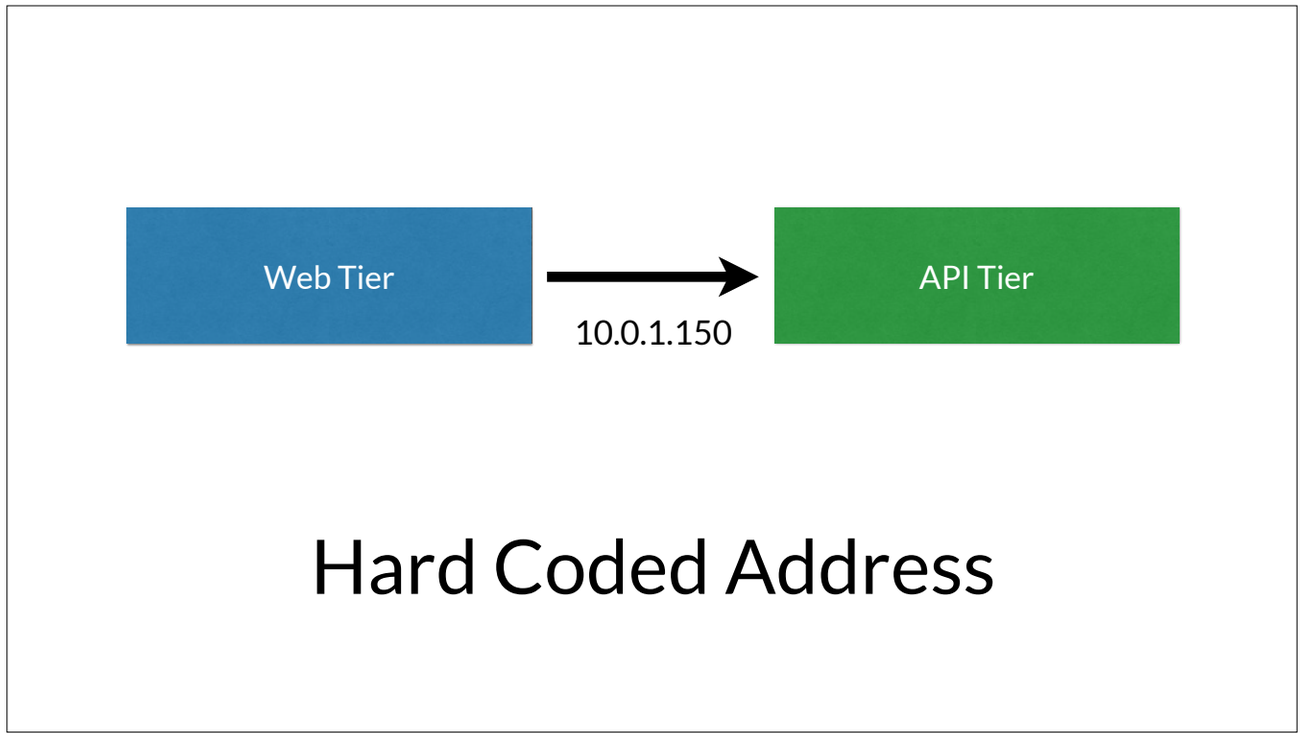
Challenges

- SOA is Hard!
- Discovery
- Monitoring
- Configuration
- Orchestration

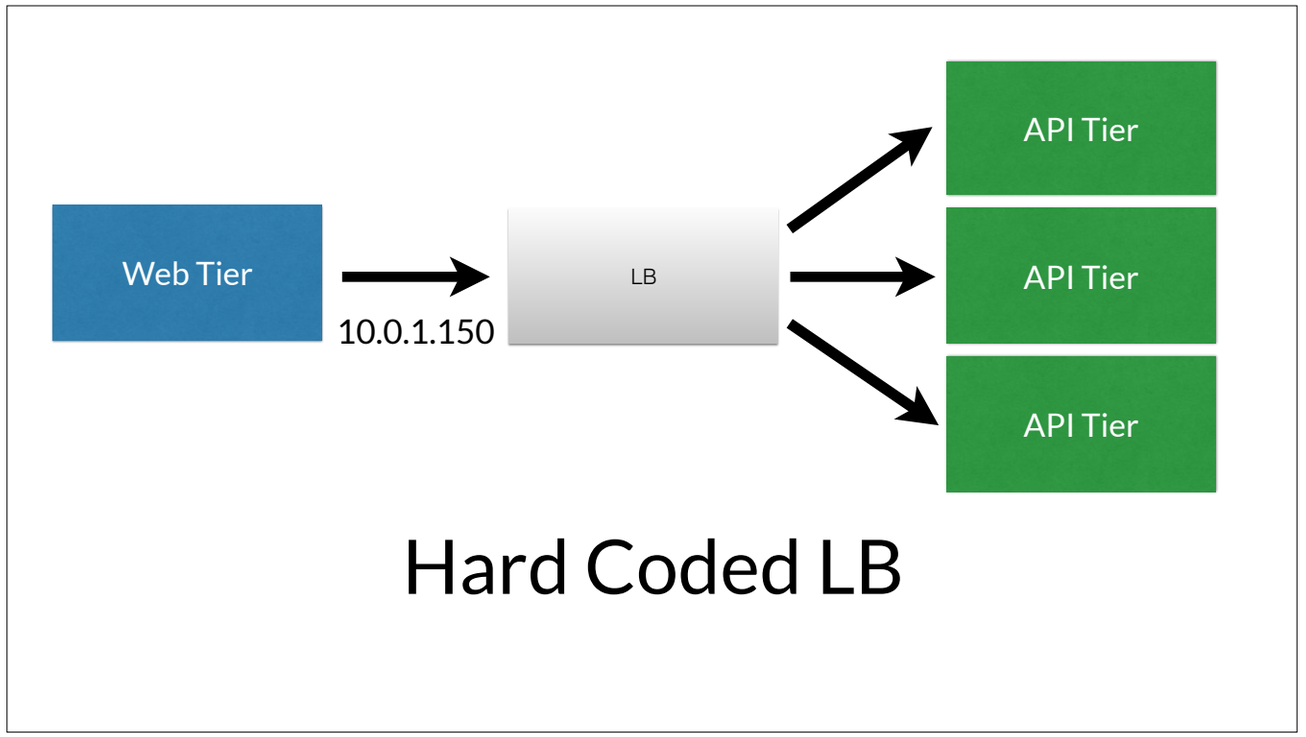
That said, SOA isn't without it's challenges. In fact, SOA is hard. There are a number of big challenges including Service Discovery, Monitoring, Configuration and Orchestration.



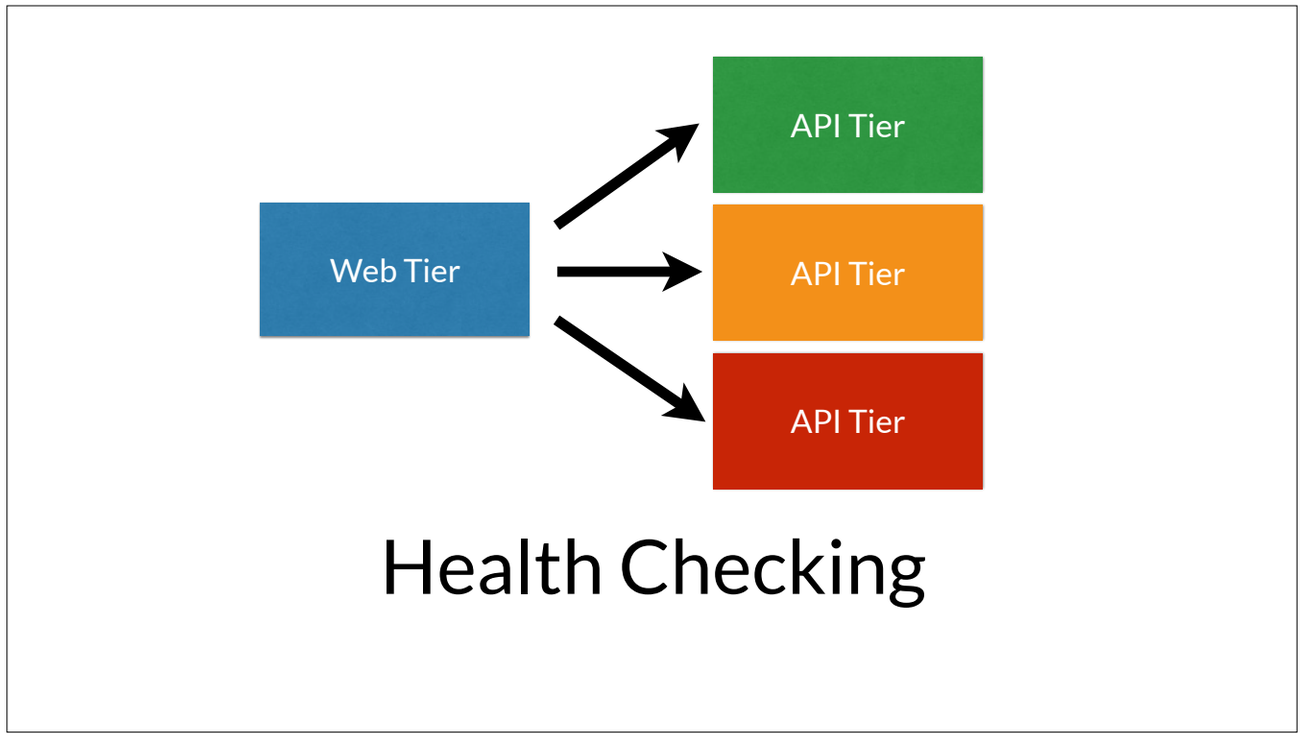
The service discovery problem is almost always the most immediate one. Given just a Web and API tier, the web servers need to be able to discover the location of the API servers.



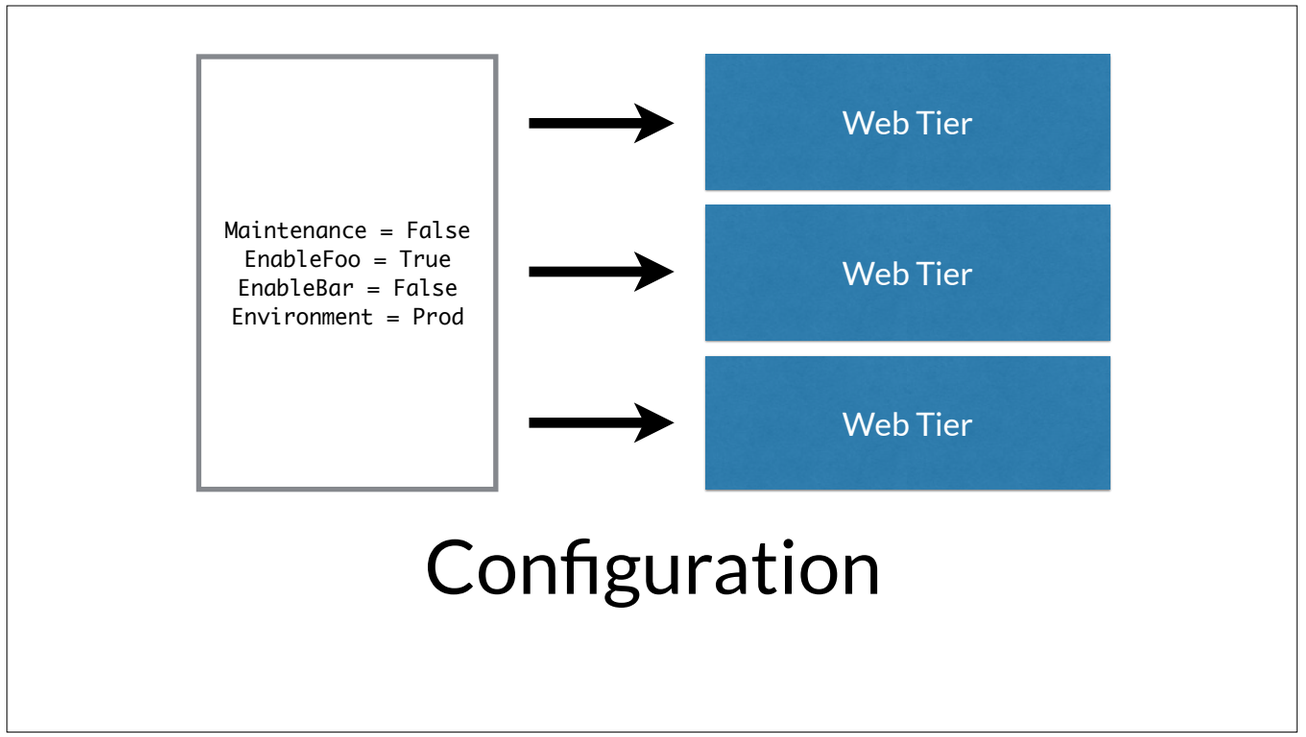
One very simple approach is to simply hard-code the address of the other tiers. This has a number of problems, it’s not a scalable approach, load balancing, and fault tolerance both are difficult. It suffers from fragility.



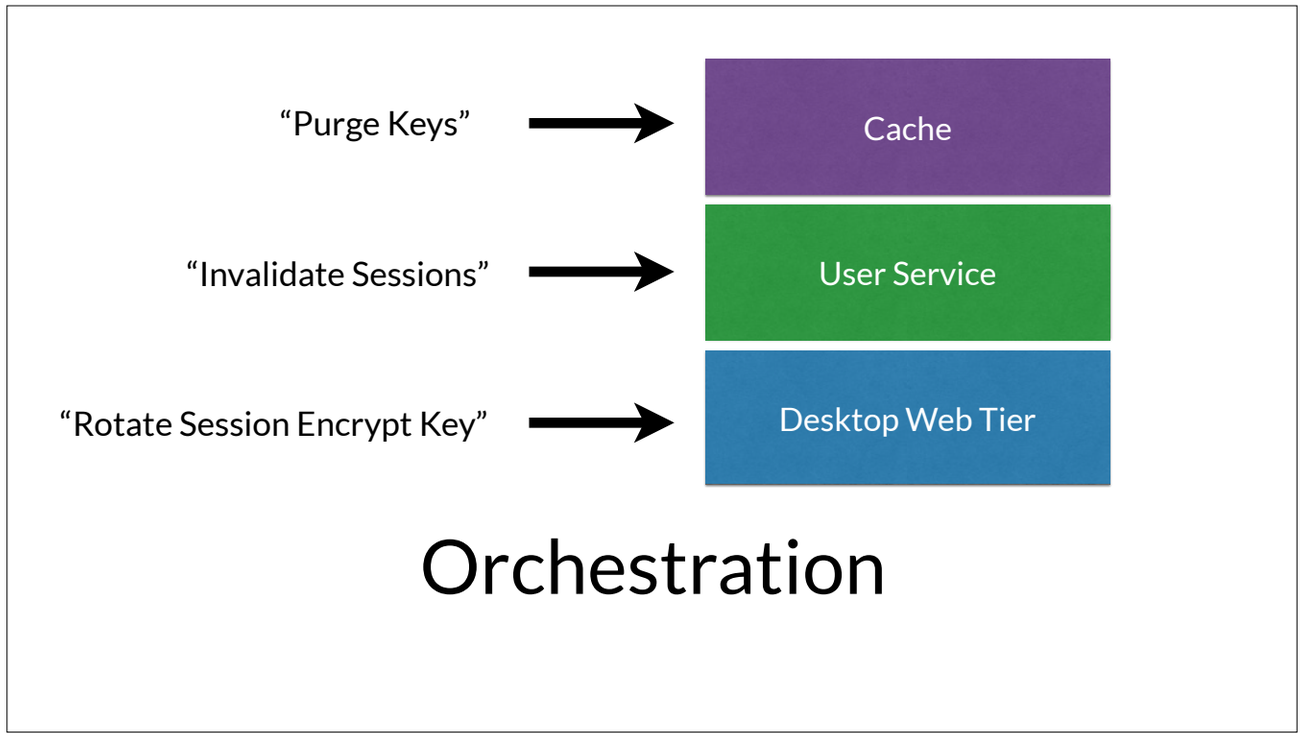
The next most common approach is the intermediate load balancer. It does allow the API tier to be scaled independently. It also allows health checking to be done by the load balancer to provide some resiliency. However, now our load balancer becomes the single point of failure.



If we are able to remove the load balancer, then we now have re-introduced the problem of health checking. Previously we could rely on the load balancer to do this for us. Now we need to both know the state of the service and also ensure traffic is not routed to any unhealthy providers.



Independent of the discovery and health checking problem is just configuration. Given that we are running a service across many machines, how do we ensure they all get timely updates to their configuration.



The last major issue is orchestration or event delivery. For many services, there are certain types of events that they are designed to respond to. Managing events and ensuring their timely delivery across a service fleet is also a challenging problem. For example, we may way to purge our caches, force user sessions to be invalidated, or maybe instruct our web tier to rotate the encryption key that is used. Maybe we are responding to yet another OpenSSL vulnerability and these need to happen ASAP.

SOA Barriers

- Discovery
- Health + Monitoring
- Configuration
- Orchestration

So clearly there are some barriers to SOA. However, putting everything in a monolith doesn't solve many of these problems either. Ultimately, none of these problems are new, they've been around for a while, and there are existing solutions.

Service Discovery

- DNS
- ZooKeeper
- RDBMS
- Redis

On the service discovery side, there are many existing tools. These are some of the prominent tools used to tackle the problem.

The problem with most of these tools is exactly that, they are tools. It’s very similar to asking your realtor for a house.

And instead being given a hammer, nails and some wood. Sure, it is entirely possible to build a service discovery solution with these components, but directions are not included.

Monitoring

- Load Balancers
- Nagios
- Sensu

Similarly, monitoring tools have existed for a while. Load Balancers usually offer some form of monitoring, and other more general purpose systems like Nagios and Sensu exist to provide global health checking.

The problem with these tools is that they silo health information. They are designed specifically to detect problems and report them to human operators. They are not designed as a feedback mechanism for service discovery. As a result, a health check may fail on a service but traffic will still be routed to it.

Configuration

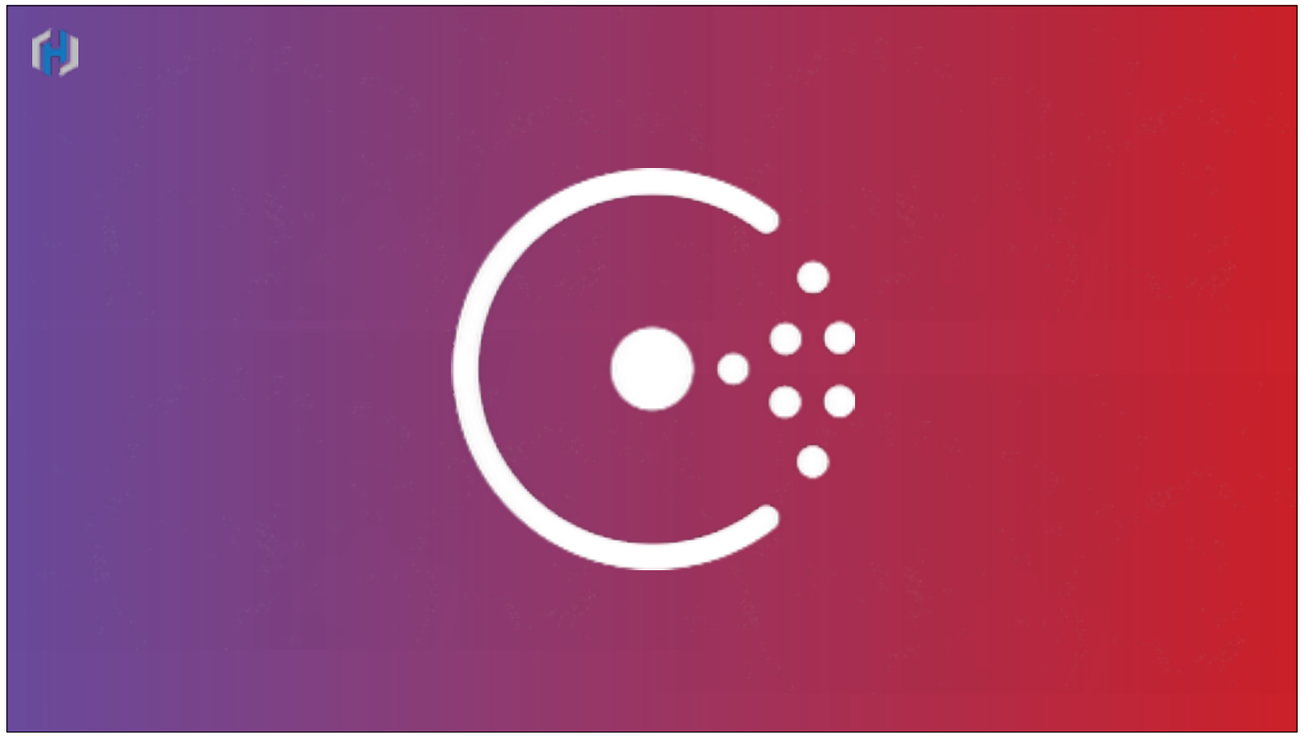
- Static Files
- Configuration Management
- RDBMS
- Redis

Configuration has probably been an issue from the moment the first application was written. This is probably the least painful problem, as most configuration tends to be relatively static. It is possible to have more dynamic configuration using config management tools, databases, redis, etc. These allow for a higher level of runtime configuration that is easy to change. Unfortunately, many of these approaches suffer from issues of speed, scalability or fault tolerance.

Orchestration

- SSH
- Redis
- ZooKeeper
- mCollective
- Custom

Orchestration is the last, but also the least solved issue. There is not broad adoption of specific tooling for this problem. Likely because it’s the easiest to ignore.



Given all of this we decided to build Consul

Consul

- Solution vs Tool
- Service Discovery
- Health Checking
- Configuration
- Orchestration

Consul is a tool, but the goal is to focus on high-level solutions. We want to give you the house and the hammer. Ultimately, it tackles all the challenges presented by a service oriented architecture.

Service Discovery

- Nodes, Services, Checks
- Simple Registration (JSON)
- DNS Interface
- HTTP API

Service discovery is at the heart of Consul. Consul uniquely blends service discovery with health checking, which is absolutely crucial. Registering services with Consul requires no code, instead a simple JSON file is provided that describes the service and optionally provides health checks. The checks are Nagios compatible, which means almost every conceivable service already has a pre-written check. Querying data out of Consul is done using either a DNS interface or the HTTP API.

```
{
    "service": {
        "name": "redis",
        "tags": ["slave"],
        "port": 8000,
        "check": {
            "script": "/usr/local/bin/check_redis.py",
            "interval": "10s"
        }
    }
}
```

As an example, here is a simple JSON registration for a redis service. We are declaring the service name, providing a tag, indicating its a ‘slave’ node, the port and lastly a health check. We’ve provided a nagios compatible check that is run on a 10 second interval to check health.

```
$ dig slave.redis.service.consul

; <<>> DiG 9.8.3-P1 <<>> slave.redis.service.consul
; (3 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 9046
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;slave.redis.service.consul. IN A

;; ANSWER SECTION:
slave.redis.service.consul. 0  IN A  10.1.10.38
```

DNS is a first-class mechanism to doing service discovery with Consul. Here we can issue a simple request to lookup the “redis” service but filtering only to instances with the “slave” tag. Consul responds with an A record providing the IP address of the redis instance. Unfortunately, that doesn’t include the port to use.

```
$ dig slave.redis.service.consul SRV

; <<>> DiG 9.8.3-P1 <<>> slave.redis.service.consul SRV
; (3 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 11480
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;slave.redis.service.consul. IN SRV

;; ANSWER SECTION:
slave.redis.service.consul. 0  IN SRV  1 1 8000 foo.node.dc1.consul.

;; ADDITIONAL SECTION:
foo.node.dc1.consul. 0  IN A  10.1.10.38
```

However, if we issue an SRV request to Consul, it will provide the appropriate records indicated both the port 8000 and the IP address of the host.

DNS Interface

- Zero Touch
- Round-Robin DNS (Randomized)
- Filters on Health Checks

Using a DNS based discovery approach allows for a zero-touch integration with Consul. No application code changes. JSON files are provided to register services, and discovery is done transparently at the DNS level.
Additionally, the DNS records returned are randomized, so you get a round-robin form of load balancing. Entries for services that are failing health checks are automatically filtered out to avoid routing to unhealthy hosts.

HTTP API

- Rich HTTP API
- Deeper integrations
- More intelligent clients

In addition to the DNS interface, there is also a rich HTTP API. The API can be used to deeper more intelligent integrations. This allow clients to customize their routing logic, connection pooling, etc. It also makes it simple to build tooling around Consul.

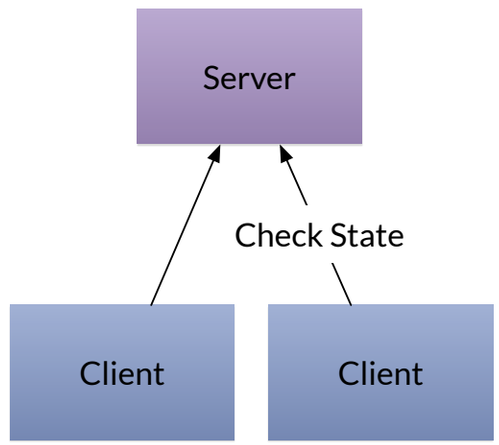
Monitoring

- Nagios Compatible
- Scalable
- DevOps friendly
- Actionable

As I mentioned, the monitoring features are deeply integrated into the service discovery of Consul. When building Consul, we wanted to ensure that the checks would be compatible with the existing nagios ecosystem, but unlike nagios we wanted the checks to be scalable, devops friendly and actionable.

Monitoring

- Run on the Nodes
- Push vs Pull
- Edge Triggered



As a result, we spent a lot of time designing this feature. We wanted to ensure Consul could scale to thousands of machines, so instead of the central servers using SSH to run the checks remotely in effect a “Pull” model, Consul runs the health checks on the edges and uses a “Push” model. This allows for a major optimization which is to only do edge triggering. If a health check is in a stable state, we don’t need to update the servers. This provides multiple orders of magnitude reduction in write volume.

Liveness

- No Heartbeats
- Gossip-based Failure Detector
- Constant Load
- Serf (http://www.serfdom.io)

The problem with edge triggered monitoring is that we have no liveness heartbeats. Meaning in the absence of any updates, we don't know if the checks are simply in a stable state or if the server has died. Consul get's around this by using a gossip-based failure detector. All cluster members take part in a background gossip, which has a constant load regardless of cluster size. This enables Consul to scale to thousands of machines with negligible load on CPU or the network.

Configuration

- Hierarchical Key/Value Store
- HTTP API
- Long-Polling
- Locking
- ACLs

To support application configuration, Consul provides a hierarchical Key/Value store. The KV store is available through a simple HTTP API. It provides support for long-polling of changes, distributed locking, and ACLs. This provides a rich set of primitives to integrate applications.

envconsul

```
$ envconsul -reload service/foo/config /usr/bin/foo
...
$ envconsul -reload=true service/foo/config env
MAX_CONNS=256
ENABLE_FOO=True
ENVIRONMENT=production
```

In practice, there are a few common ways of integrating application configuration with Consul. One approach is a tool called ‘envconsul’. Inspired by ‘envdir’, it simply reads the keys from the KV store at a given prefix, and sets them as environmental variables for the application. It can optionally reload the application on configuration changes, allowing real-time changes to be made.

consul-template

- Dynamically renders templates
- Reload on changes
- Integrate with everything*
- Real-time

Another approach is to use a tool like consul-template. This allows templates to be rendered using dynamic values read from the KV store. consul-template also handles watching for any changes, so that the config file can be re-rendered and the application reloaded, allowing for the same realtime response to changes.

consul-template

```
backend frontend
  maxconn {{ key "frontend/maxconn" }}
  balance roundrobin{{range service "app.frontend"}}
  service {{.ID}} {{.Address}}:{{.Port}}{{end}}
```

```
backend frontend
  maxconn 256
  balance roundrobin
  server web1 10.0.1.100:80
  server web2 10.0.2.200:80
```

consul-template also integrates with both the KV store and the service catalog. This lets us do things like populate an HAProxy configuration with variables like the maximum connection count from the KV store, and the list of healthy servers from the service catalog. All of this is without HAProxy needing to be aware of Consul in any way, consul-template just glues the two together.

Orchestration

- Watchers
- Event System
- Remote Execution
- Distributed Locking

When it comes to Orchestration, Consul provides an entire toolkit of features. All of these features “just work” out of the box, so that developers and operators don’t have to write code to get started.

Watchers

- Watch for Changes
- “View of Data”
- Custom Handlers
- KV, Services, Nodes, Checks, Events
- Static or Dynamic

Watches are the simplest way to react to changes using Consul. A watcher is basically a pairing between a “view of data” and a custom handler. The “data” you care about can be KV data, services, nodes, health checks, or custom events. The handler can be any executable, letting operators customize behavior. Watches can be configured either statically using configuration or with the “consul watch” command dynamically.

Watchers

```
$ consul watch -type nodes
[
    {
        "Node": "foo",
        "Address": "10.1.10.38"
    }
]
$ consul watch -type nodes /usr/bin/setup_dns.sh
```

As an example, a watcher can be setup to watch for changes to the nodes list. Any time a new node joins or leaves the cluster, the handler will be invoked. If we invoke it without a handler, we just get the current result set. We can also specify a handler which gets updates over stdin and is free to react in any way. For example, we could update our BIND configuration.

Events

- Flexible event system
- Name + Payload
- Filter on Services, Nodes, Tags
- Built on Watches

Building on watches, Consul includes a flexible event system that can be used for any purpose. Events are fired with a name and payload, and handlers can be registered to respond to events. Events can filter on services, nodes and tags, so that you can target the machines that should process the event. Because events are built on watches, they operate in the same way. Either statically configured or using consul watch.

Events

```
$ consul event -service redis -name test sample-payload
Event ID: 8cf8a891-b08b-6286-8f68-44c6930cafe0
$ consul watch -type event -name test cat
[{“ID”:”8cf8a891-b08b-6286-8f68-44c6930cafe0”,
“Name":"test",
“Payload":"c2FtcGxlLXBheWxvYWQ=",
“NodeFilter":"",
“ServiceFilter":"redis",
“TagFilter":"",
“Version":1,
"LTime":4}]
```

As an example, here we are firing an event named “test”, targeting the “redis” service and providing a “sample-payload”. If we had been running a watcher, it could filter to only “test” events. Here we can see the handler receives the event, with the base64 encoded payload.

Remote Execution

- Built on Event System
- Parallel SSH++
- Filter on Services, Nodes, Tags

In addition to the generic event system, Consul also provides a remote execution tool. This allows operators to invoke commands like parallel SSH, while filtering on services, tags, or nodes. This can be used to avoid writing a specific event handler when a more generic remote exec is suitable.

Remote Execution

```
$ consul exec -service redis uptime
    foo: 18:24  up 79 days,  1:10, 1 user, load averages: 10.72 5.34 4.78
    foo:
==> foo: finished with exit code 0
1 / 1 node(s) completed / acknowledged
```

This is a simple example of running the “uptime” command against any node that is providing the “redis” service. The event system is used to notify nodes of the remote execution, and then the request and all the response data is sent back using the KV store. This avoids an expensive parallel SSH to all the nodes, and allows for very efficient execution.

Distributed Locking

- Built into KV
- Client-side Leader Election
- Semaphores

Lastly, to support more complex orchestration needs, Consul supports distributed locking. This is supported within the Key/Value store and allows applications to implement client-side leader election or semaphores. This requires a deeper integration with Consul, but frees developers from needing to develop their own consensus protocols.

High-Level Features

- Service Discovery
- Health Checking
- Configuration
- Orchestration

We've covered briefly some of the high-level features of Consul. The point was not to comprehensively cover the details of all of these. The documentation does a much better job of that, we've just hit the tip of the iceburg.

Solutions vs Tools

- Opinionated Solutions
- 95% Use Case
- “It Just Works”
- House, not the hammer

Instead, the point was to demonstrate that Consul provides a set of opinionated solutions to these common problems. The goal is not to solve every use case, but to cover the 95% and to do it with a “it just works” experience. We want to provide the house, and not just the hammer.

Datacenter Control Plane

In this way, we actually think of Consul as a datacenter control plane.

Control Plane

- “Batteries Included”
- Scales to thousands of machines
- Multi-Datacenter Support
- Infrastructure Backbone

As a control plane layer, it comes with all the batteries included. It is designed (and deployed) at thousand node scale, and it supports multiple datacenters. It provides the backbone on which the rest of infrastructure is built.

Deployment

- Agent per Machine
- Server and Client Agents
- 3-5 servers per datacenter

We've talked about the *What* and *Why* with Consul, but not the *How*. Consul requires the use of an agent, which runs on every machine. The agent can be in either a client or server mode. Each datacenter must have at least one server, but usually either 3 or 5 are deployed for fault tolerance.

From a 10,000 ft view, Consul looks like this. Here we have two datacenters, containing client and server nodes. Within each datacenter, all of the nodes are participating in a LAN gossip. The server nodes have a single node elected leader and are replicating to the others. Requests make use of RPC forwarding, so a client forwards to a server, which then forwards to the leader node. Alternatively, a request may be forwarded to a remote datacenter as well. The servers participate in a second level of WAN gossip allowing data centers to be aware of each other.

Agents

- RPC, HTTP, DNS APIs
- Health Checks
- Event Execution
- Gossip Pool

All of the agents, regardless of being in a client or server mode provide the same set of RPC, HTTP, and DNS APIs. As a client of Consul, the difference is transparent. The queries are appropriately forwarded to a server, either in the local or a remote datacenter if appropriate. All the agents run health checks and handle even execution locally and push results to the servers as appropriate.

Gossip

- Based on Serf (http://www.serfdom.io)
- Membership
- Failure Detection
- Event System

The Gossip pool the agents participate in is based on Serf. Serf can be used an independent Cli or embedded as a library like in Consuls case. It provides membership, failure detection and an event system.

Gossip Membership

- Join Requires Any Peer
- Server Discovery
- Protocol Versioning

The gossip membership features are great for operators. They mean only 1 other member of the cluster needs to be known to join. This allows clients to automatically find all the servers in the cluster without any configuration. Lastly, it provides a layer to do protocol versioning. This allows Consul to maintain a very strong level of backwards compatibility.

Failure Detector

- Leaving vs Failing
- Automatic Reaping
- Distributed Locking with Liveness
- No Heartbeating
- Constant Load

In addition to membership, the gossip layer provides a distributed failure detector. This provides a number of useful features. It allows us to distinguish between a node leaving the cluster and failing. This allows us to reap nodes from the service catalog automatically. It also lets us build distributed locking in a way that provides liveness, meaning progress can be made in the face of failure. The distributed nature of the detector means that we also don’t use centralized heartbeating and have a constant load on the servers regardless of cluster size.

Gossip Events

- P2P delivery
- <1s delivery for 1K+ nodes
- No server load

Lastly, the gossip layer provides a very efficient event system. Because messages are delivered peer-to-peer, we get sub-second delivery on clusters with thousands of nodes. It also avoids placing load on central servers to manage delivery and broadcast of events.

Server Agents

- Centralized state
- Query Handling
- Raft Consensus
- WAN Gossip

The server nodes are the “brains” of the cluster. They maintain the state, and replicate data between themselves. They are also responsible for responding to queries. As a detail, they make use of the Raft Consensus protocol, which is a simplification of Paxos. They also participate in a WAN gossip pool, this means the servers are aware of the other data centers and can forward requests as appropriate.

System Service

- Ubiquitous Agents
- Unified API
- Treat as System Service
- Useful for Operators and Developers

Because of the design of Consul, the agent ends up being ubiquitously deployed. Additionally, since the API is unified across agent types, from a user perspective we are guarded from the internal details of Consuls implementation. In this way, you can think of Consul as a system service, that can be used by system operators and developers.

Consul in Production

- 2K+ stars (GitHub)
- 100K+ downloads
- VagrantCloud
- Large-scale customer deploys

To really see the impact of Consul, it's useful to see cases of its use in production. The popularity of Consul is really quite staggering. It has over 2K stars on GitHub, and over 100K downloads. It is the backbone of HashiCorp's VagrantCloud infrastructure, which I'll talk more about. It also has seen some very large-scale customer deploys as well.

Vagrant Cloud

- SaaS Product for Vagrant
- Vagrant Share
- Box Discovery
- Box Hosting
- Service Oriented Architecture

Vagrant Cloud is a SaaS service for Vagrant. It powers the “share” feature of Vagrant,
provides box discovery and hosting. This includes the actual VM images and the associated metadata. I believe it recently crossed the 2M+ mark for box downloads. All of it is built around a service oriented architecture build on Consul.

Consul in Vagrant Cloud

- Single AWS VPC
- Dozens of Services
  - Internal + External
- 3 Consul Servers

Vagrant Cloud is deployed within a single VPC on AWS. It is composed of dozens of services. Some of these are internal and hosted by us, and others are hosted externally using SaaS providers.

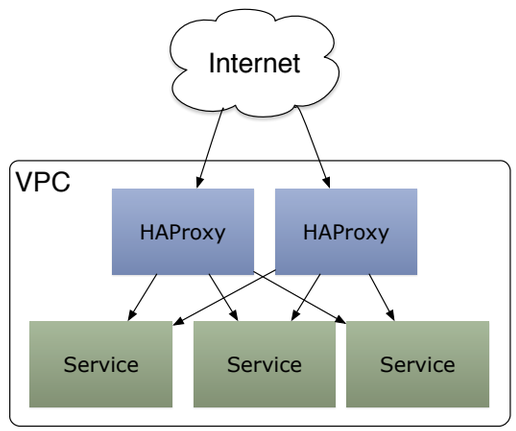
Discovery in Vagrant Cloud

- DNS for most Services
- Query Agent for Local Service
  - Binary Streaming
- HAProxy at Edges

To handle service discovery in Vagrant Cloud, we mostly rely on DNS. There are some exceptions to this. Some of our services query the local agent to check for a possible co-process. For example, when doing a binary upload we would prefer to talk to a service on localhost instead of going over the network. The edges of our network also relies on HAProxy.

HAProxy

- Consul Internal Only
- HAProxy “Bridge”
- consul-template

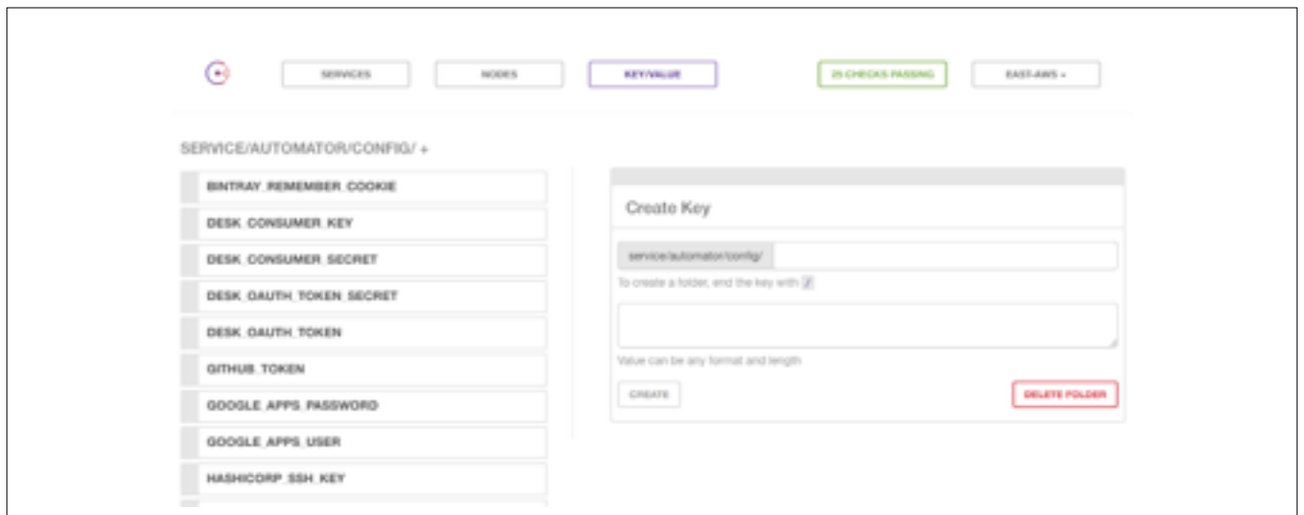


One reason to use HAProxy is that Consul is an internal facing service. There is no simple way to have an external DNS service query Consul, and even then, the addresses are generally private and not-routable. HAProxy provides a “bridge” to the outside world. To manage this setup, we simply use consul-template to keep the HAProxy configuration up-to-date.

Application Configuration

- consul-template
- envconsul

We also use consul-template more generally to configure internal applications. Many of our simpler services can read all of their configuration from environmental variables, so we can use envconsul for that.



Consul Web UI

To manage setting and updating the configuration, we simply use the Consul Web UI. The UI is included as part of the releases, and provides an easy way to browse services, nodes, health checks, and to manipulate the KV store.

Vagrant Cloud Orchestration

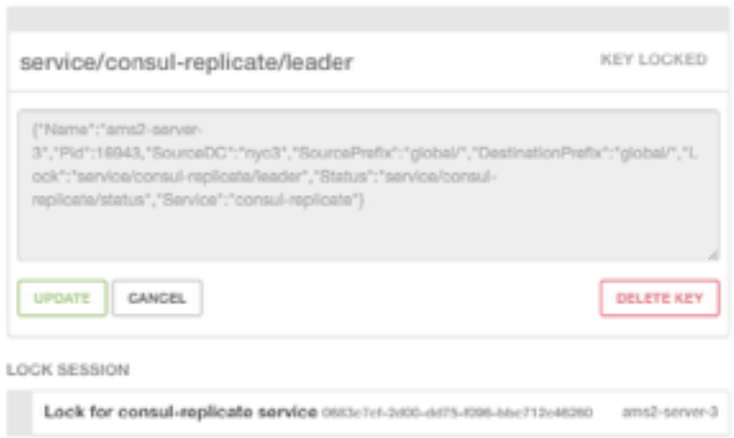
- Deploys
- Leader Election
- Service Coordination (StorageLocker)

But arguably Consul has ingrained itself most deeply in how we orchestrate Vagrant Cloud. It is used to manage deploys, do leader election within services, and provide deeper coordination. One example I'll talk about is our use of it in StorageLocker.

Deploys

- Event Driven
- Download Binaries
- Rolling Restarts

One very useful integration with Consul is for application deploys. We simply wire specific events for various services, and use this to trigger downloading the new binaries and performing a rolling restart of the service. This makes updates as simple as issuing a single “consul exec”.



Client-Side Leader Election

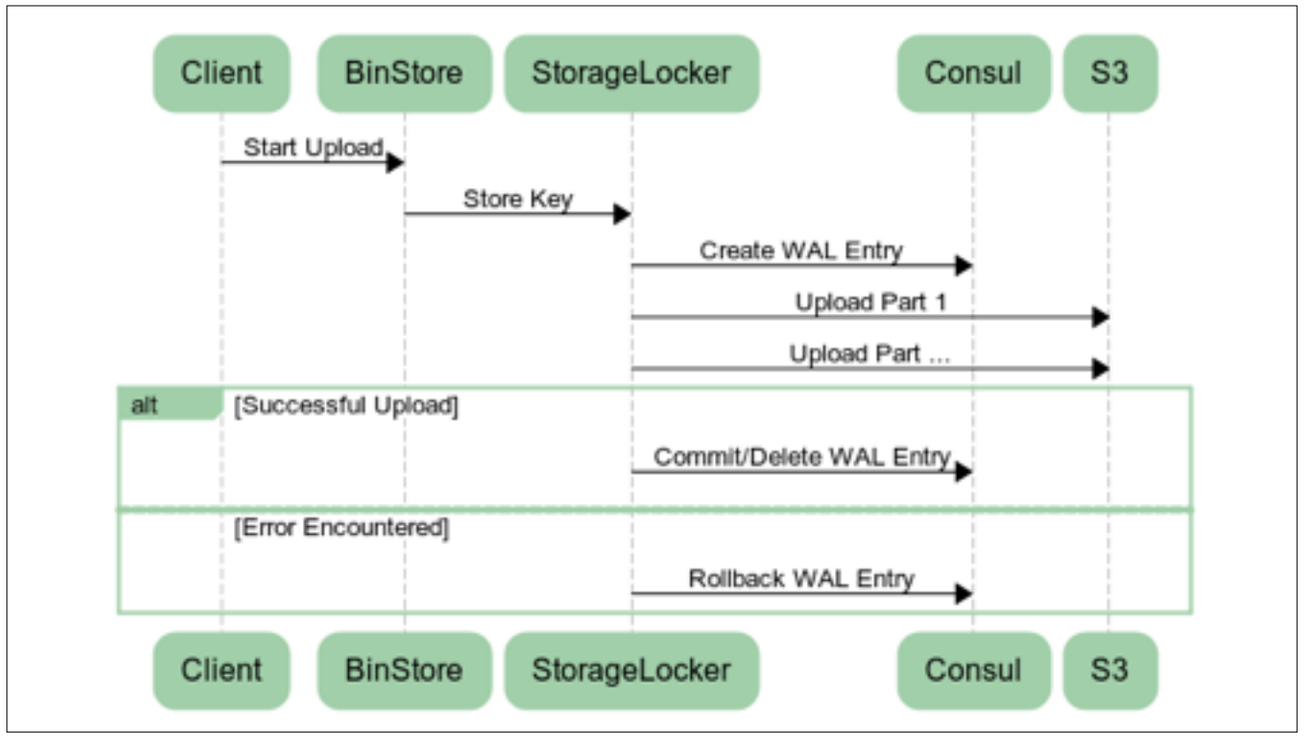
Highly Available Services, N+1 Deploys

It also provides an essential building block for doing client-side leader election. This allows us to have highly available services. Instead of deploying a single instance of a service, we can deploy N+1 and use the leader election to ensure only one instances is running. Best of all, Consul gives us insight into details like who is the current leader very easily.

Service Coordination

- StorageLocker, BLOB Storage
- Write-Ahead-Logs
- Atomic Multi-Part Uploads

Lastly, one of my favorite uses of Consul internally is for a service we call StorageLocker. It is a relatively simple service, it provides BLOB Storage with a simple Put, Get, Delete API. StorageLocker protects against partial failures by using a Write Ahead Log. The WAL logs are stored in Consul and tied to its Session and Health Checking. This allows us to get atomic multi part uploads to backends like S3.



We have a blog post that covers how this works in more detail, but at a very high level, we create a WAL entry in Consul before uploading or deleting files from the storage backend. We can either commit the WAL entry on success or we can abort and rollback. If the StorageLocker instance handling the request dies, the other instances can detect this using Consul and automatically start the rollback. Using Consul in this way didn't add much complexity but dramatically increased the robustness of the service.

Consul + Vagrant Cloud

- Simplified Operations
- More Robust Applications
- Realtime Reactions
- Minimum Ops Pain!

In summary, using Consul with Vagrant Cloud has simplified our ops efforts. Building our applications in a Consul aware manner has allowed us to make more robust applications by using the available primitives instead of rolling them into the app. We get real-time responsiveness to changes in configuration or deployed instance, and best of all we've suffered no ops pain. It just works.

Industry Use

- Many Industry Deploys
- Power Tail
- Few “Very Large”
- Several “Large”
- Many “Medium”

In addition to powering Vagrant Cloud, Consul is seeing deployments in Industry as well. It’s hard to give exact usage numbers, but we’ve had over 100K+ downloads of Consul so far. Of course most deployments aren’t large, but we work with a few customers who have “Large” and “Very Large” Deploys.

Lithium Networks

- SaaS, Fortune 500 Customers
- Hybrid Cloud
  - Physical, OpenStack, AWS

One of the “large” deploys of Consul is by a very large SaaS company, Lithium Networks. They have a complex hybrid cloud infrastructure that spans physical hardware, OpenStack and AWS.

Deployment Stats

- 1000+ Nodes
- 4 Datacenters
- 3 Environments (QA, Stage, Prod)
- Service Discovery
- 15% Fleet Reduction

This particular deployment has over 1000 nodes. This spans 4 datacenters, with 3 different regions per data center. They are primarily using Consul for Service Discovery, but they've already reduced their fleet size by 15% by just removing intermediary load balancers.

CDN Company

- “Very Large” Deployment
- 2000+ Machines
- 12+ POPs

Another “Very Large” deployment comes from a CDN Company. Unfortunately we are also still waiting on legal to release their name. They operate at a massive scale, with over 2000 machines and over 12 points-of-presense, each which operates as a datacenter.

Consul @ CDN Company

- Replacing ZooKeeper
- DNS + HAProxy Discovery
- Application Configs in KV, not CM
- Client-Side Election for Orchestration
- Move Fast, Less Ops

These guys are aggressively adopting Consul and taking advantage of the features it provides. They are replacing their ZooKeeper usage. In doing so, they are adopting DNS and HAProxy for service discovery internally. Application configuration is moving from config management tools. They are also using Consul to manage client-side leader election in their services. For them, Consul enables them to move faster and do less ops.

Over the Horizon

- Power Tail
- Many deploys with 100’s of Servers
- Next order of magnitude

This is just two anecdotes. There are lots of other deploys of various size, but most follow a power tail. There are many more deploys in the hundreds of nodes scale as well. At the same time, we are actively working with some customers on testing Consul with an order of magnitude large clusters than we’ve seen so far. Hopefully we’ll get to talk about them at the next conference!

Common Challenges

- Infrastructure is a means to an ends
- Service Discovery
- Monitoring
- Configuration
- Orchestration

One thing I want to point out is that with Vagrant Cloud, and all our industry users, Infrastructure is a means to an ends. The value is in the application layer. Simultaneously, almost all of us are faced with the same challenges. We all need to have service discovery, monitoring, configuration and orchestration. However, until now we've lacked good tooling to do this.

Patchwork Solutions

- Re-invent the wheel
- Done Right vs Done Fast
- Maintenance
- Minimum Viable vs Maximum Utility

Instead what we see is a patch work of different systems being glued together. This involves a lot of re-inventing the wheel, tradeoffs between doing it something correctly or quickly. It also burdens an organization with maintaining their solution. As a result of this, most solutions are of the minimum viable as opposed to providing the most utility possible.

Instead, we propose Consul as the alternative. It provides a single answer for all of the runtime problems in a datacenter, and can act as the single control plane.

Consul

- “Batteries Included”
- Works at Scale
- Robust + Reactive Infrastructure

Consul comes with all the batteries included. You don’t need to figure out how to piece everything together, it just works out of the box. It is also designed to work at scale, so you can deploy it with a dozen servers and scale to thousands without worrying about redesigning your entire architecture. It also allows you to build a more robust and reactive infrastructure as a whole, reducing operations pain and allowing you to move faster.

Thanks!

Q/A