9
Antipatterns - Avoiding Counterproductive Solutions

We have looked at a number of good processes, practices, and patterns for the AWS cloud. Now, we will explore some that you want to avoid. Some of the examples that follow are going to be countered by using the items in the first two sections. Others are not inherently technology problems. You should be aware of the smells that they bring and change course when you run into them. As your cloud craftsmanship improves, they will be easier to spot before implementation. At the beginning of your cloud journey, we expect failure to occur all the time. Be aware that refactoring is always going to be an option moving past your minimum viable product. Often, speed is the most important dimension to your product success. Taking on technical debt may be required to increase market shares or drive new features.

We will briefly cover the following topics in this chapter:

- Exploring counterproductive processes
- Practices to avoid in general
- Anti-patterns that you might come across

Exploring counterproductive processes

The first category we will explore is counterproductive processes. The cloud provides a whole new set of features that you can take advantage of. Migrating existing processes to your AWS-based product will work in most cases, but you will miss out on a great opportunity to make things better. Doing things the same way in the cloud will lead you back to where you are today. Consider how you add the greatest value for your customer before you spend time repeating what you did before the cloud existed.

The sections that follow cover processes that can be used, but probably shouldn't.

Lift and shift

Wholesale migration of your existing systems, products, practices, and patterns to AWS is our first anti-pattern. You can do it, but seriously think about it. If your only objective is cost savings and you have no plans to ever change your product offerings, go for it. However, you will end up loading even more technical debt onto your teams. This will make it even harder to change course when the market eventually shifts away from what you are doing. Small experiments with cloud migration will be a better solution while lessening your systemic risk.

Change control boards

If you have a complicated change review process, it's recommended that you reduce or remove this process before you move to AWS. Following test-driven, peer reviewed development practices for your product provides additional benefits without all the bureaucracy typically found in a **change control board** (**CCB**). In conjunction with desilofication (covered later in this chapter), the removal of this step will increase your time-to-value immensely.

That's not to say that removing the CCB is going to be an easy task. CCB's are usually in place to provide companies with a feeling of security and the ability to manage risk – sometimes, deployments are seen as just plain risky.

Non-reproducibility

Manual processes are inherently risky. We have all typed in the wrong password or forgotten to close a quote. Let your automation take care of as much as possible. If you find yourself doing the same thing more than twice, seriously consider automating it. The whole industry is moving in the direction of idempotence (the property of certain operations that allows users to repeatedly make the same call without producing any change to the initial result). The microservice and serverless patterns give us the ability to minimize variation within our systems.

Firefighting

Remember all that technical debt you created by ignoring the previous three smells? At some point, things will start exploding (if they're not burning already). If you find yourself trying to contain a tire fire, your product will suffer the most. All the time you could have dedicated to delighting your users will be wasted dousing flare-ups. The next section will cover some of the data we can use to speed up the smothering. Breathing in some smoke occasionally is fine. Not ever seeing which direction you are going will asphyxiate your product.

Can't fail attitude to system uptime

Being in a place where your product cannot fail is good if lives depend on your system. Any place else, you need to expect breakdowns. If you are not running a mission-critical system and people are telling you that failure is not an option, it might be time to find another job. The number of combinations you can put together with AWS services, coupled with duplication to ensure availability, integrity, and confidentiality increases the odds of a deficiency. Chaotic engineering experiments can build confidence in a system. Concise service level agreements help build consumer confidence and manage customer expectations about possible downtime.

Practices to avoid in general

There is going to be some overlap here with the previous section, but we will focus on behaviors at a higher level. As you move to the AWS cloud, you have an excuse to review your traditional methods to ensure they are still applicable. In most cases, the technology is not the largest inhibiting factor. A lack of shared understanding will be the biggest blocker.

The following sections cover some of the most common practices that should be avoided.

Silos

Conway's law (`http://www.melconway.com/Home/Conways_Law.html`) suggests that an organization is bound to create software that mimics its structure. Think about this as you move your workloads to the cloud. There is an opportunity to refactor your team, as well as your systems and software. Organizational design is out of the scope of this book, but we will touch on some patterns to aid you in implementing good development practices later in the chapter.

The outcome of those exercises will provide input to the desilofication that should be done before you get too far in your cloud journey. An easy win in this analysis is usually to be found in your testing and quality assurance areas. As we discussed earlier, moving these processes forward in your development process will bear rewards in the velocity and quality spaces. Involving your security teams straightaway will also lessen your process waste.

Lock in

Organizational composition can also force you into sub-optimal patterns. Discussions with the same old people can get you stuck in a rut that it is difficult to clamber out of. Out-of-the-box thinking needs to be encouraged, as there is no box except the one you put yourself into. On a similar note, try to avoid solutions with a single option. Try to choose technologies that adhere to well-known standards and offer a pool of competing alternatives.

Amazon offers a great many operating systems from which to choose for your instances. EKS is built on Kubernetes, which is a mature project maintained by a well-established foundation (`https://www.cncf.io/projects/`). Open source roots also give you the ability to contribute back to the community and protect yourself from forced retirement of a component.

Version control

Keeping track of components such as code, configurations, and credentials is crucial. There are many people who will tell you that everyone should commit to master, while others espouse the benefits of branches. We will not get into that. If you do not keep old revisions of everything, it will catch up with you. Components must be handled in a way that allows for retrieval of previous iterations. The ability to mix new options of one with the other during testing is crucial to continual experimentation and long-term cloud success.

Anti-patterns that you might come across

AWS is awesome. Companies like Hacker News, Reddit, and Instagram could never have succeeded without its utility form. The challenge is not to recreate your current problems in the cloud. For startups who are building from scratch, it is easy to fall into the same traps that have historically slowed innovation and diminished security. The evolution of micro-services occurred because dependency management and release coordination are difficult problems to solve over the course of a lifetime for a large product.

The following sections cover some anti-patterns, usually used in traditional deployment environments, that can be translated into AWS environments.

Monoliths

You certainly can paint yourself into a corner with AWS. In order to speed your product to market, you will build a **big ball of mud** (`https://en.wikipedia.org/wiki/Big_ball_of_mud`). Even large companies can find themselves in a mess with many services being created, but with little forward-planning. As a counter to this, organizations like Amazon and Netflix use **microservices** – small services tailored to do only one thing, but to do it well.

The Rise of Microservices website (`https://www.appcentrica.com/the-rise-of-microservices/`) details more about how microservices came about, how and why they are used, and how to handle them in production.

At some point, you will be able to take a breath and address all the technical debt you have accumulated, or everything will blow up at a very inconvenient time. Recognizing when to refactor is key. Using explicit techniques such as bounded context from the **Domain-Driven Design** world (`https://vaughnvernon.co/`) will help you decompose your monolith into microservices. Be advised though: as with everything there are trade-offs. Managing several microservices, let alone many hundreds or even thousands of microservices, is, in itself, a complicated task.

Single points of failure

Netflix has run into all kinds of issues on AWS. Estimates suggest that one-quarter of the traffic on the internet is theirs. Amazon is always improving their products in order to delight their customers, but also to eliminate single points of failure. Global customers provide the insight to build more reliable services while maintaining availability and low cost. Common occurrences of **single points of failure** in non-cloud environments were usually found in the connectivity realm – or rather, networking (see the following subsection). Other elements that can be seen as single points of failure are data stores and web-service hosting environments/application servers, where only single instances are used. Having environments without redundancy is usually a sign of bad design.

Networking

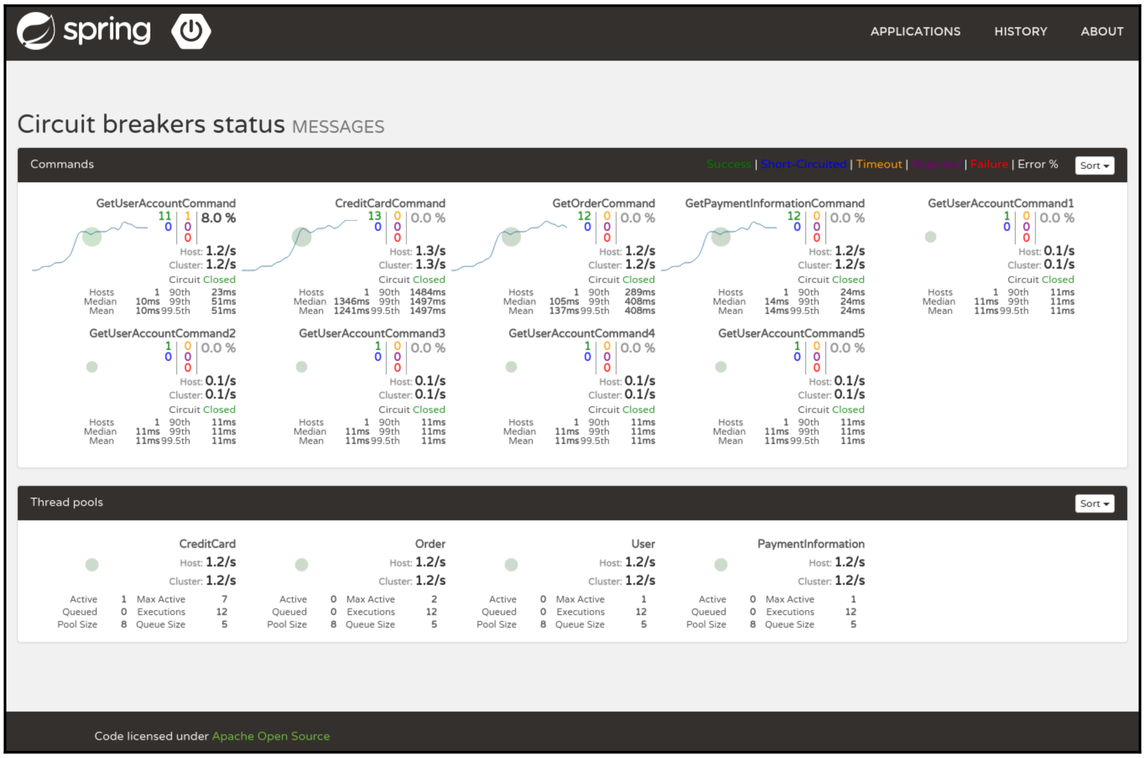
One key benefit of the Amazon Cloud is its abundance of addresses. In the past, the **Internet Protocol** (**IP**) version 4 furnished us with a large address space. Now that there are billions of mobile phones on the planet, we are running out of headroom. **Network Address Translation** (**NAT**) supplied us with a workaround to the challenge. Although NAT will keep us going for years to come, IPv6 is the future. Each person on the planet can be assigned trillions of addresses at a time without exhausting the pool. Consider using IPv6 for your internal networks moving forward to eliminate complex NATing schemes. The flexibility to change easily outweighs the burden of NAT. If you must use NAT, ensure that your DNS, load balancers, and instances can be quickly recreated in a new region (the low-cost option). For critical components, ensure availability through fault-tolerant designs. Don't forget about your gateways, networks, and routes.

Scaling

Netflix ran into more classes of failures as they scaled. They propagated feedback loops that took their own systems offline. These downstream effects were mitigated by some new patterns lifted from traditional engineering disciplines. First, they began using caches to reduce to load on postliminary systems. Eventually, up-to-date systems were good enough for many of their products. Next, came bulkheads. This construct allows developers to mirror the divide provided by availability zones and regions in their software.

Resilience

The following screenshot shows the Hystrix (`https://github.com/Netflix/Hystrix`) dashboard for a Spring Boot (`http://spring.io/projects/spring-boot`) application. The circuit breaker was one of the first patterns Netflix put into place to help product owners provide graceful degradation in a consistent manner:



Circuit breakers status MESSAGES

```
Commands                      Success | Short-Circuited | Timeout | Rejected | Failure | Error %

GetUserAccountCommand         11 | 1 | 8.0 %     Host: 1.2/s   Cluster: 1.2/s   Circuit Closed
                               0 | 0
                                 | 0
  Hosts   1       90th    23ms
  Median  10ms    99th    51ms
  Mean    10ms  99.5th    51ms

CreditCardCommand             13 | 0 | 0.0 %     Host: 1.3/s   Cluster: 1.3/s   Circuit Closed
                               0 | 0
                                 | 0
  Hosts   1       90th    1484ms
  Median  1346ms  99th    1497ms
  Mean    1241ms 99.5th   1497ms

GetOrderCommand               12 | 0 | 0.0 %     Host: 1.2/s   Cluster: 1.2/s   Circuit Closed
                               0 | 0
                                 | 0
  Hosts   1       90th    289ms
  Median  105ms   99th    408ms
  Mean    137ms  99.5th   408ms

GetPaymentInformationCommand  12 | 0 | 0.0 %     Host: 1.2/s   Cluster: 1.2/s   Circuit Closed
                               0 | 0
                                 | 0
  Hosts   1       90th    24ms
  Median  14ms    99th    24ms
  Mean    14ms  99.5th    24ms

GetUserAccountCommand1         1 | 0 | 0.0 %     Host: 0.1/s   Cluster: 0.1/s   Circuit Closed
                               0 | 0
                                 | 0
  Hosts   1       90th    11ms
  Median  11ms    99th    11ms
  Mean    11ms  99.5th    11ms

GetUserAccountCommand2         1 | 0 | 0.0 %     Host: 0.1/s   Cluster: 0.1/s   Circuit Closed
                               0 | 0
                                 | 0
  Hosts   1       90th    11ms
  Median  11ms    99th    11ms
  Mean    11ms  99.5th    11ms

GetUserAccountCommand3         1 | 0 | 0.0 %     Host: 0.1/s   Cluster: 0.1/s   Circuit Closed
                               0 | 0
                                 | 0
  Hosts   1       90th    11ms
  Median  11ms    99th    11ms
  Mean    11ms  99.5th    11ms

GetUserAccountCommand4         1 | 0 | 0.0 %     Host: 0.1/s   Cluster: 0.1/s   Circuit Closed
                               0 | 0
                                 | 0
  Hosts   1       90th    11ms
  Median  11ms    99th    11ms
  Mean    11ms  99.5th    11ms

GetUserAccountCommand5         1 | 0 | 0.0 %     Host: 0.1/s   Cluster: 0.1/s   Circuit Closed
                               0 | 0
                                 | 0
  Hosts   1       90th    11ms
  Median  11ms    99th    11ms
  Mean    11ms  99.5th    11ms

Thread pools

CreditCard             Host: 1.2/s   Cluster: 1.2/s
  Active     1  Max Active    7
  Queued     0  Executions   12
  Pool Size  8  Queue Size    5

Order                  Host: 1.2/s   Cluster: 1.2/s
  Active     0  Max Active    2
  Queued     0  Executions   12
  Pool Size  8  Queue Size    5

User                   Host: 1.2/s   Cluster: 1.2/s
  Active     0  Max Active    1
  Queued     0  Executions   12
  Pool Size  8  Queue Size    5

PaymentInformation     Host: 1.2/s   Cluster: 1.2/s
  Active     0  Max Active    1
  Queued     0  Executions   12
  Pool Size  8  Queue Size    5
```

Code licensed under Apache Open Source

The preceding screenshot can be found at `https://github.com/VanRoy/spring-cloud-dashboard/blob/master/screenshot-circuit-breaker.png`.

When building resilient systems, timeouts are notoriously difficult to manage. Both upstream and downstream dependencies can provide their own settings, and managing the aggregates can be a delicate operation. By allowing for a valid response code in the event of insufficient capacity, Netflix products exhibited graceful degradation in the event of any timeout. Amazon adopted many of these models for their own retail properties. Rate limiting, bulkheading, automatic retries, and response caching patterns equip your product developers with tried and true methods to respond to cloud-scale traffic bursts.

Summary

We have looked at solutions that are ineffective and may result in undesired consequences. Avoiding these cloud anti-patterns promotes increased quality and lower defects throughout your cloud adventure. We have covered how to eschew recreating organization barriers in AWS, recognize unsuitable cloud workloads, and regularly detect opportunities for improvement. In the next chapter, we will identify how to collect and learn from data that can help us avoid anomalies that haven't been widely encountered in the cloud yet.

Further reading

- *AWS Networking Essential* (`https://subscription.packtpub.com/video/virtualization_and_cloud/9781788299190`).
- Refer to `Chapter 1`, *Breaking the Monolith* for bounded context, and `Chapter 4`, *Client Patterns* and `Chapter 5`, *Reliability Patterns* for patterns of the *Microservices Development Cookbook* (`https://subscription.packtpub.com/book/application_development/9781788479509`) book.
- *DevOps with Git* (`https://subscription.packtpub.com/video/application_development/9781789618839`).
- *resilience4j* (`https://github.com/resilience4j/resilience4j`) delivers Java exemplars you can reuse in your code.
- Envoy (`https://www.envoyproxy.io/`) is a container-based solution that can centralize many of the configuration responsibilities of developers.