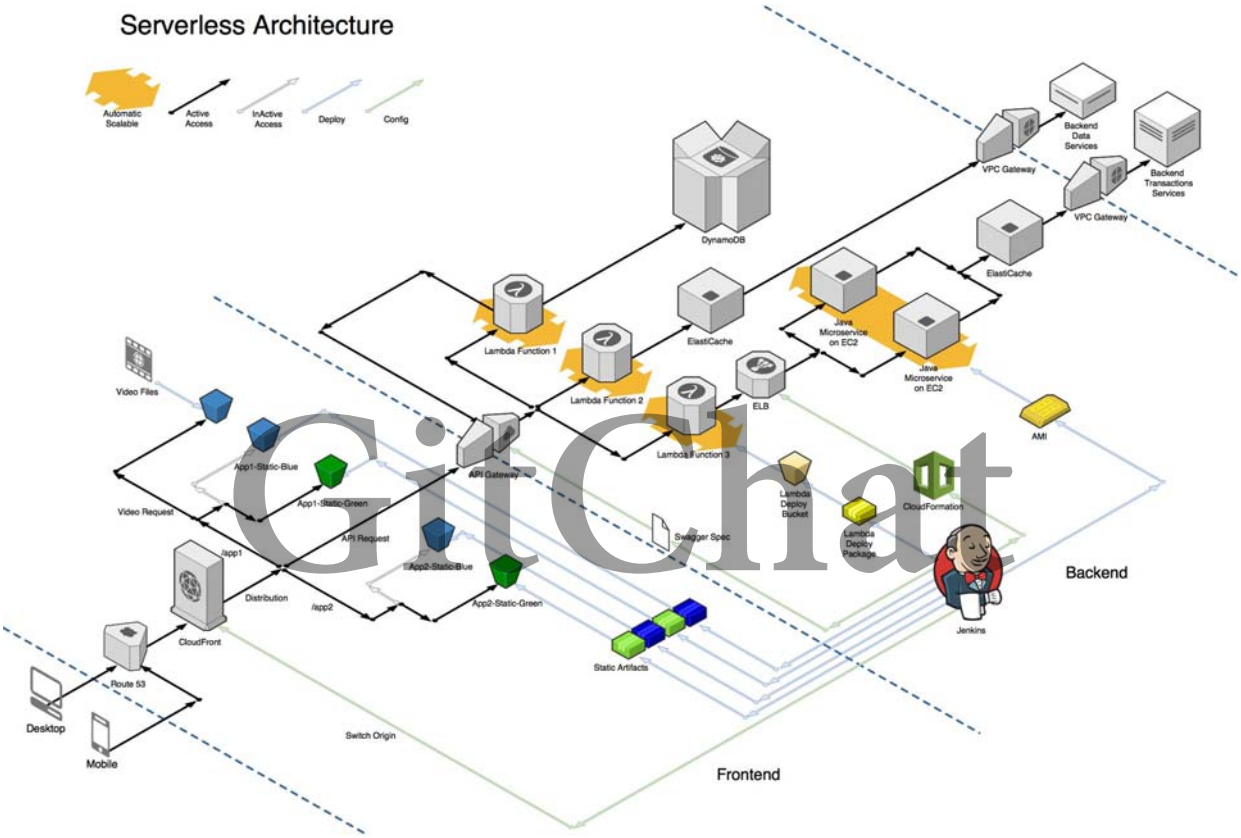
Serverless 微服务的持续交付

“Serverless 风格微服务的持续交付（上）：架构案例”中，我们介绍了一个无服务器风格的微服务的架构案例。这个案例中混合了各种风格的微服务，本文介绍 Serverless 风格的微服务部分的持续交付。

架构图如下：



在这个架构中，我们采用了前后端分离的技术。我们把 HTML，JS， CSS 等静态内容部署在 S3 上，并通过 CloudFront 作为 CDN 构成了整个架构的前端部分。我们把 Amazon API Gateway 作为后端的整体接口连接后端的各种风格的微服务，无论是运行在 Lambda 上的函数，还是运行在 EC2 上的 Java 微服务，他们整体构成了这个应用的后端部分。

从这个架构图上我们可以明显的看到 前端（Frontend）和后端（Backend）的区分。

持续部署流水线的设计和实现

任何 DevOps 部署流水线都可以分为三个阶段：待测试，待发布，已发布。

由于我们的架构是前后端分离的，因此我们为前端和后端分别构造了两条流水线，使得前后端开发可以独立。如下图所示：

在这种情况下，前端团队和后端团队是两个不同的团队，可以独立开发和部署，但在发布的时候则有些不同。由于用户是最后感知功能变化的。因此，为了避免界面报错找不到接口，在新增功能的场景下，后端先发布，前端后发布。在删除功能的场景下，前端先发布，后端后发布。

我们采用 Jenkins 构建我们的流水线，Jenkins 中已经含有足够的 AWS 插件可以帮助我们完成整个端到端的持续交付流水线。

前端流水线

前端持续交付流水线如下所示：

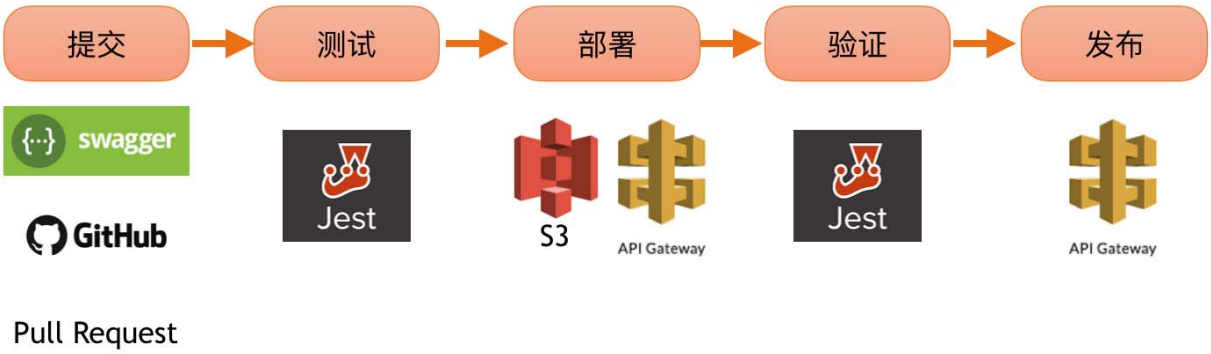
前端流水线的各步骤过程如下：

1. 我们采用 BDD/ATDD 的方式进行前端开发。用 NightWatch.JS 框架做 端到端的测试，mocha 和 chai 用于做某些逻辑的验证。
2. 我们采用单代码库主干（develop 分支）进行开发，用 master 分支作为生产环境的部署。生产环境的发布则是通过 Pull Request 合并的。在合并前，我们会合并提交。
3. 前端采用 Webpack 进行构建，形成前端的交付产物。在构建之前，先进行一次全局测试。
4. 由于 S3 不光可以作为对象存储服务，也可以作为一个高可用、高性能而且成本低廉的静态 Web 服务器。所以我们的前端静态内容存储在 S3 上。每一次部署都会在 S3 上以 build 号形成一个新的目录，然后把 Webpack 构建出来的文件存储进去。
5. 我们采用 Cloudfront 作为 CDN，这样可以和 S3 相互集成。只需要把 S3 作为 CDN 的源，在发布时修改对应发布的目录就可以了。

由于我们做到了前后端分离。因此前端的数据和业务请求会通过 Ajax 的方式请求后端的 Rest API，而这个 Rest API 是由 Amazon API Gateway 通过 Swagger 配置生成的。前端只需要知道 这个 API Gateway，而无需知道API Gateway 的对应实现。

后端流水线

后端持续交付流水线如下所示：



后端流水线的各步骤过程如下：

1. 我们采用“消费者驱动的契约测试”进行开发，先根据前端的 API 调用构建出相应的 Swagger API 规范文件和示例数据。然后，把这个规范上传至 AWS API Gateway，AWS API Gateway 会根据这个文件生成对应的 REST API。前端的小伙伴就可以依据这个进行开发了。

2. 之后我们再根据数据的规范和要求编写后端的 Lambda 函数。我们采用 NodeJS 作为 Lambda 函数的开发语言。并采用 Jest 作为 Lambda 的 TDD 测试框架。

3. 和前端一样，对于后端我们也采用单代码库主干（develop 分支）进行开发，用 master 分支作为生产环境的部署。

4. 由于 AWS Lambda 函数需要打包到 S3 上才能进行部署，所以我们先把对应的构建产物存储在 S3 上，然后再部署 Lambda 函数。

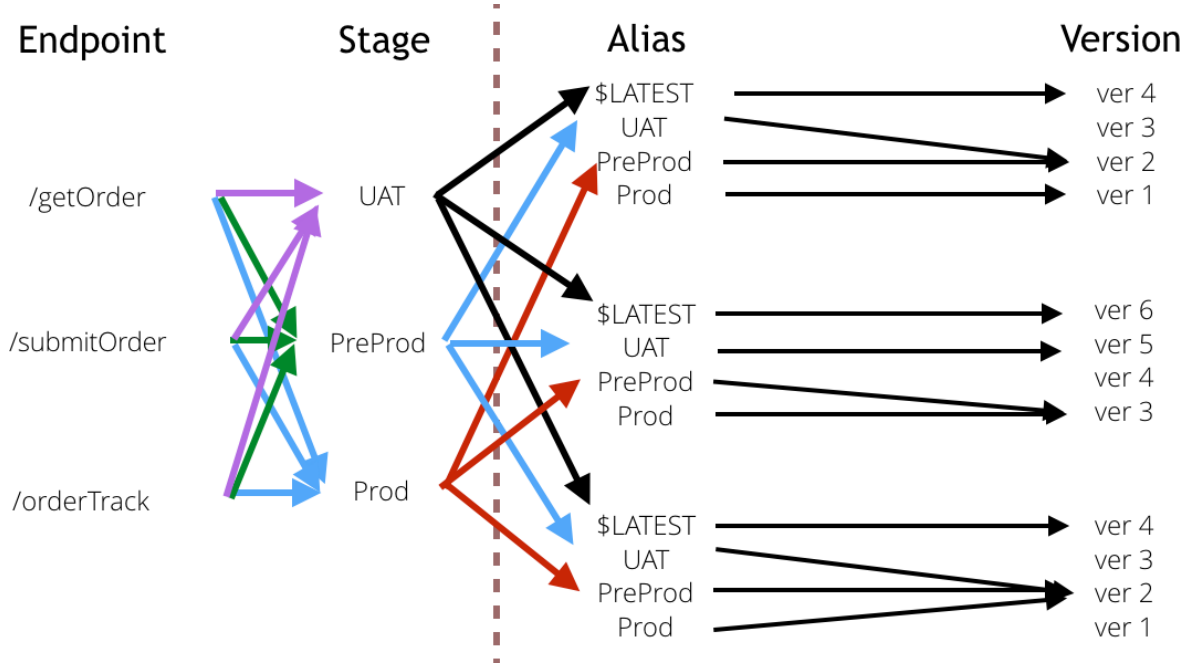
5. 我们采用版本化 Lambda 部署，部署后 Lambda 函数不会覆盖已有的函数，而是生成新版本的函数。然后通过别名（Alias）区分不同前端所对应的函数版本。默认的 $LATEST，表示最新部署的函数。此外我们还创建了 Prod，PreProd, uat 三个别名，用于区分不同的环境。这三个别名分别指向函数某一个发布版本。例如：函数 func 我部署了4次，那么 func 就有 4个版本（从1开始）。然后，函数 func 的 $LATEST 别名指向 4 版本。别名 PreProd 和 UAT 指向 3 版本，别名 Prod 在 2 版本。

6. 技术而 API 的部署则是修改 API Gateway 的配置，使其绑定到对应版本的函数上去。由于 API Gateway 支持多阶段（Stage）的配置，我们可以采用和别名匹配的阶段绑定不同的函数。

7. 完成了 API Gateway 和 Lamdba 的绑定之后，还需要进行一轮端到端的测试以保证 API 输入输出正确。

8. 测试完毕后，再修改 API Gateway 的生产环境配置就可以了。

部署的效果如下所示：



无服务器微服务的持续交付新挑战

在实现以上的持续交付流水线的时候，我们踩了很多坑。但经过我们的反思，我们发现是云计算颠覆了我们很多的认识，当云计算把某些成本降低到趋近于 0 时。我们发现了以下几个新的挑战：

1. 如果你要 Stub，有可能你走错了路。
2. 测试金子塔的倒置。
3. 你不再需要多个运行环境，你需要一个多阶段的生产环境 (Multi-Stage Production)。
4. 函数的管理和 NanoService 反模式。

Stub？别逗了

很多开发者最初都想在本地建立一套开发环境。由于 AWS 多半是通过 API 或者 CloudFormation 操作，因此开发者在本地开发的时候对于AWS 的外部依赖进行打桩（Stub）进行测试，例如集成DynamoDB（一种 NoSQL 数据库），当然你也可以运行本地版的 DynamoDB，但组织自动化测试的额外代价极高。然而随着微服务和函数规模的增加，这种管理打桩和构造打桩的虚拟云资源的代价会越来越大，但收效却没有提升。另一方面，往往需要修改几行代码立即生效的事情，却要执行很长时间的测试和部署流程，这个性价比并不是很高。

这时我们意识到一件事：如果某一个环节代价过大，你需要思考一下这个环节存在的必要性。

由于 AWS 提供了很好的配置隔离机制，于是为了得到更快速的反馈，我们放弃了 Stub 或构建本地 DynamoDB，而是直接部署在 AWS 上进行集成测试。只在本地执行单元测试，由于单元测试是 NodeJS 的函数，所以非常好测试。

另外一方面，我们发现了一个有趣的事实，那就是：

测试金子塔的倒置

由于我们采用 ATDD 进行开发，然后不断向下进行分解。在统计最后的测试代码和测试工作量的的时候，我们有了很有趣的发现：

End-2-End （UI）的测试代码占30%左右，占用了开发人员 30% 的时间（以小时作为单位）开发和测试。

集成测试（函数、服务和 API Gateway 的集成）代码占 45%左右，占用了开发人员60% 的时间（以小时作为单位）开发和测试。

单元测试的测试代码占 25%左右，占用了10%左右的时间开发和测试。

一开始我们以为我们走入了”蛋筒冰激凌反模式“或者”纸杯蛋糕反模式“（请见 http://www.51testing.com/html/57/n-3714757.html）但实际上：

1. 我们并没有太多的手动测试，绝大部分自动化。除了验证手机端的部署以外，几乎没有手工测试工作量。
2. 我们的自动化测试都是必要的，且没有重复。
3. 我们的单元测试足够，且不需要增加单元测试。

但为什么会造成这样的结果呢，经过我们分析。是由于 AWS 供了很多功能组件，而这些组件你无需在单元测试中验证（减少了很多 Stub 或者 Mock），只有通过集成测试的方式才能进行验证。因此，**Serverless** 基础设施大大降低了单元测试的投入，但把这些不同的组件组合起来则劳时费力。如果你有多套不一致的环境，那你的持续交付流水线配置则是很困难的。因此我们意识到：

你不再需要多个运行环境，你只需要一个多阶段的生产环境 (Multi-Stage Production)

通常情况下，我们会有多个运行环境，分别面对不同的人群：

1. 面向开发者的本地开发环境
2. 面向测试者的集成环境或测试环境（Test，QA 或 SIT）
3. 面向业务部门的测试环境（UAT 环境）
4. 面向最终用户的生产环境（Production 环境）

然而多个环境带来的最大问题是环境基础配置的不一致性。加之应用部署的不一致性。带来了很多不可重现问题。在 DevOps 运动，特别是基础设施即代码实践的推广下，这

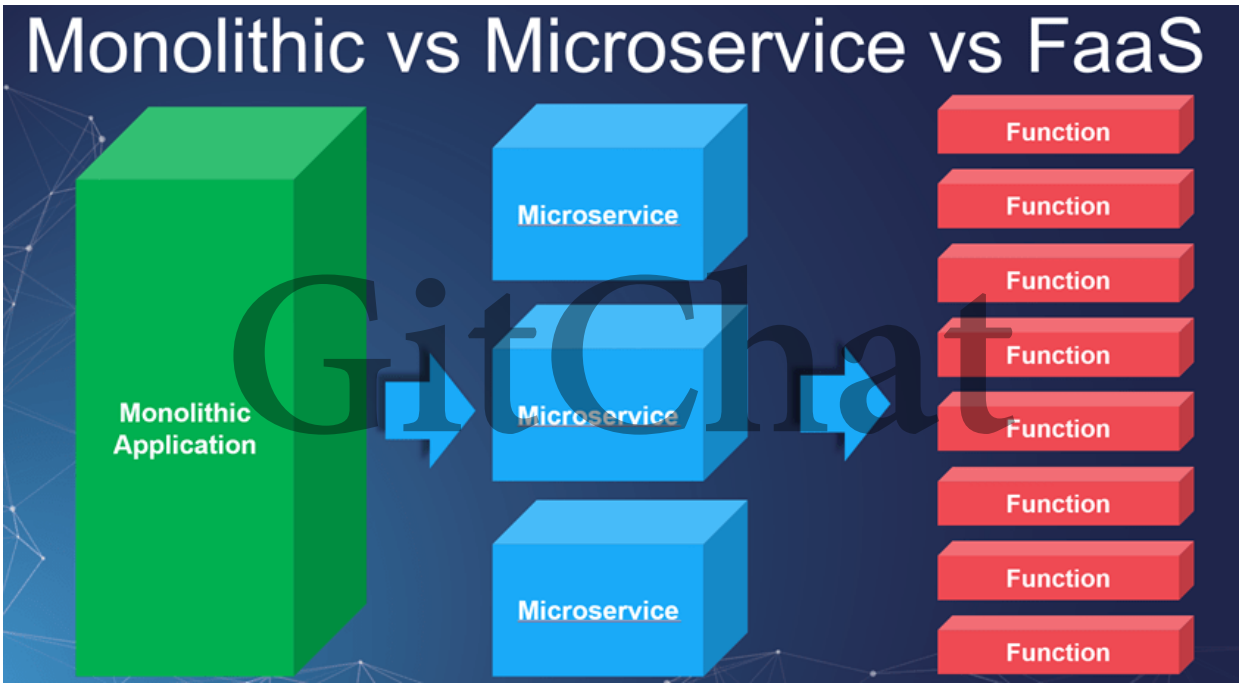
一问题得到了暂时的缓解。然而无服务器架构则把基础设施即代码推向了极致：只要能做到配置隔离和部署权限隔离，资源也可以做到同样的隔离效果。

我们通过 DNS 配置指向了同一个的 API Gateway，这个 API Gateway 有着不同的 Stage：我们只有开发（Dev）和生产（Prod）两套配置，只需修改配置以及对应 API 所指向的函数版本就可完成部署和发布。

然而，多个函数的多版本管理增加了操作复杂性和配置性，使得整个持续交付流水线多了很多认为操作导致持续交付并不高效。于是我们在思考

对函数的管理和” NanoServices 发模式 “

根据微服务的定义，AWS API Gateway 和 Lambda 的组合确实满足 微服务的特征，这看起来很美好。就像下图一样：



但当Lambda 函数多了，管理众多的函数的发布就成为了很高的一件事。而且， 可能会变成” NanoService 反模式“（https://www.infoq.com/news/2014/05/nano-services）：

> Nanoservice is an antipattern where a service is too fine-grained. A nanoservice is a service whose overhead (communications, maintenance, and so on) outweighs its utility.

如何把握微服务的粒度和函数的数量，就变成了一个新的问题。而 Serverless Framework (https://serverless.com/)，就是解决这样的问题的。它认为微服务是由一个多个函数和相关的资源所组成。因此，才满足了微服务可独立部署可独立服务的属性。它把微服务当做一个用于管理 Lambda 的单元。所有的 Lambda 要按照微服务的要求来组织。Serverless Framework 包含了三个部分：

1. 一个 CLI 工具，用于创建和部署微服务。

2. 一个配置文件，用于管理和配置 AWS 微服务所需要的所有资源。
3. 一套函数模板，用于让你快速启动微服务的开发。

此外，这个工具由 AWS 自身推广，所以兼容性很好。但是，我们得到了 Serverless 的众多好处，却难以摆脱对 AWS 的依赖。因为 AWS 的这一套架构是和别的云平台不兼容的。

所以，这就又是一个“自由的代价”的问题。

CloudNative 的持续交付

在实施 Serverless 的微服务期间，发生了一件我认为十分有意义的事情。我们客户想增加一个很小的需求。我和两个客户方的开发人员，客户的开发经理，以及客户业务部门的两个人要实现一个需求。当时我们 6 个人在会议室里面讨论了两个小时。讨论两个小时之后我们不光和业务部门定下来了需求（这点多么不容易），与此同时我们的前后端代码已经写完了，而且发布到了生产环境并通过了业务部门的测试。由于客户内部流程的关系，我们仅需要一个生产环境发布的批准，就可以完成新需求的对外发布！

在这个过程中，由于我们没有太多的环境要准备，并且和业务部门共同制定了验收标准并完成了自动化测试的编写。这全得益于 Serverless 相关技术带来的便利性。

我相信在未来的环境，如果这个架构，如果在线 IDE 技术成熟的话（由于 Lambda 控制了代码的规模，因此在线 IDE 足够），那我们可以大量缩短我们需求确定之后到我功能上线的整体时间。

通过以上的经历，我发现了 CloudNative 持续交付的几个重点：

1. 优先采用 SaaS 化的服务而不是自己搭建持续交付流水线。
2. 开发是离不开基础设施配置工作的。
3. 状态和过程分离，把状态通过版本化的方式保存到配置管理工具中。

而在这种环境下，Ops工作就只剩下三件事：

1. 设计整体的架构，除了基础设施的架构以外，还要关注应用架构。以及优先采用的 SaaS 服务解决问题。
2. 严格管理配置和权限并构建一个快速交付的持续交付流程。
3. 监控生产环境。

剩下的事情，就全部交给云平台去做。