自动实现无需干预的安全部署

Clare Liguori

自动实现无需干预的安全部署

当我参加 Amazon 工作面试时，我着重问了其中一个面试官：“你们部署到生产环境的频率如何?” 当时，我正在开发一个产品，这个产品每年推出一次或两次主要版本，但是有时候我需要在主要版本之间发布小型修补程序。 对于所发布的每个修补程序，我都要花费几个小时来仔细推出。然后，我疯狂地查看日志和指标，看看产品在部署后是否破坏了任何内容，是否需要回滚。

我之前读到 Amazon 会实施持续部署，所以，在我接受面试时，我想要了解作为 Amazon 的开发人员，我需要花多少时间来管理和监控部署。 面试官告诉我，更改会通过持续部署管道每天多次自动部署到生产环境。当我问他每天花多少时间仔细管理每项部署，并像我那样密切监控日志和指标，以查看是否会有任何影响时，他告诉我通常不花任何时间。因为管道会为他的团队完成这些工作，大部分部署都不需要任何人主动监控。 “哇！” 我说。我在加入 Amazon 后，发现这些“无需干预”的自动化部署是多么地令人兴奋。

安全的持续部署如何为开发人员腾出时间

此后，我亲眼看到 Amazon 如何通过设置持续部署管道来帮助我们快速安全地进行部署。对于我们的持续部署安全实践如何让开发人员从部署工作中解脱出来，我十分赞赏。 当我将生产代码推入服务的源代码存储库的主要分支中时，我通常会忘记这件事并继续我的下一项任务，与此同时，我团队的管道会替我将相关更改发布到生产环境中。 将代码更改发布到生产服务完全是由管道自动完成的，这意味着上一次我或任何其他开发人员修改或查看代码的同时，代码更改便会合并到源代码存储库。

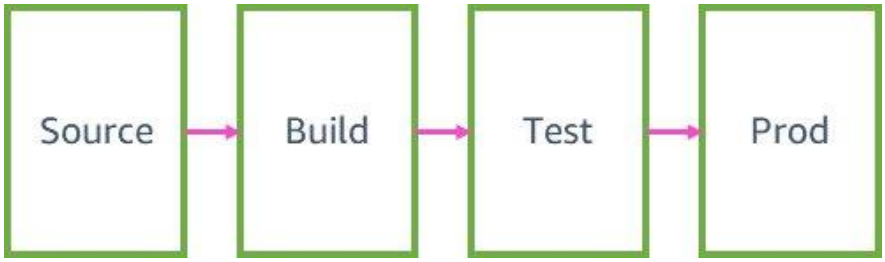
我的团队在管道中设置了自动步骤，可将我们的更改安全地部署到生产环境，因此，我们无需监控每项部署。管道通过一组测试和部署安全检查来运行最新更改。这些自动化步骤可以防止影响客户的缺陷到达生产环境，即使这些缺陷到达了生产环境，也能限制对客户带来的影响。 作为开发人员，我能够相信这个管道将谨慎并且安全地为我将更改部署到生产环境，而无需我主动去监控。

持续交付之旅

Amazon 当时并没有开始实施持续交付，这里的开发人员以前要用几小时到几天的时间来管理将代码部署到生产环境的过程。 我们在整个公司采用了持续交付的方式，以自动化、标准化我们部署软件的方式，并减少将更改部署到生产环境所用的时间。对我们发布流程的改进会随着时间的推移逐步增加。我们确定了部署风险，并找到了通过管道中的新安全自动化措施来缓解这些风险的方法。我们通过识别新风险，并找到提高部署安全性的新方法，继续迭代发布流程。 要了解关于我们的持续交付之旅，以及我们如何持续改进的更多信息，请参阅 Builders' Library 文章采用持续交付，加速交付进度。

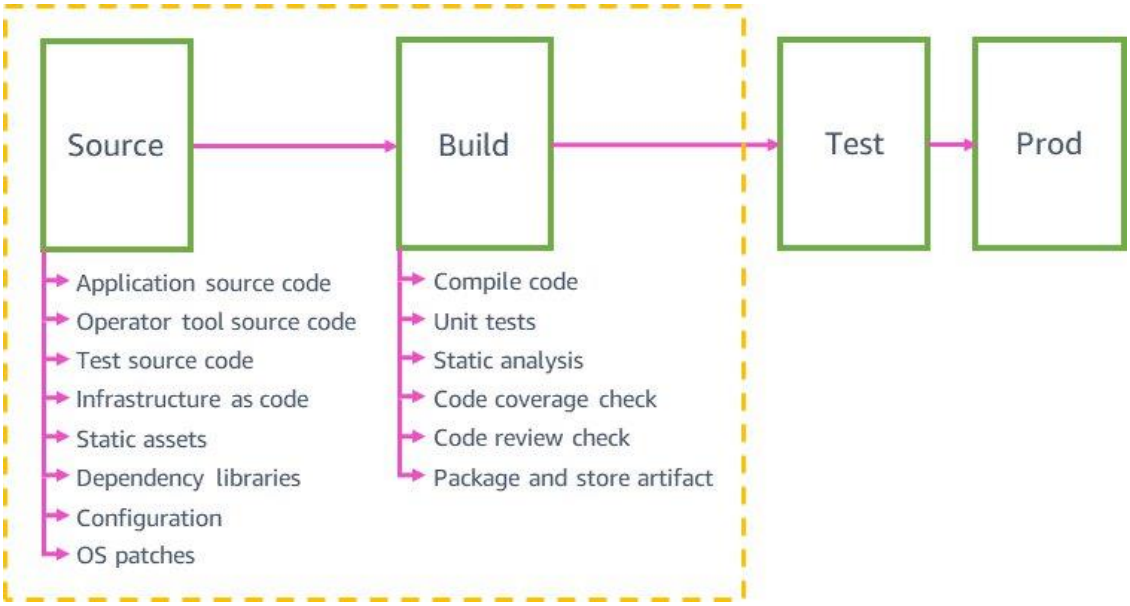
管道的四个阶段

在这篇文章中，我们将演示代码更改通过 Amazon 管道到达生产环境的步骤。 典型的持续交付管道包括四个主要阶段 – 源、构建、测试和生产。 我们将详细讲述典型 AWS 服务在各个管道阶段中的详细内容，并为您提供示例，介绍一个典型 AWS 服务团队如何设置他们的一个管道。



源和构建

下图简要介绍了您在典型 AWS 服务团队管道中可能会发现的源和构建步骤。



管道源

Amazon 管道会自动验证任何类型的源更改，并将其安全地部署到生产环境，不仅仅是对应用程序代码的更改。 它们可以验证和部署对源的更改，例如网站静态资产、工具、测试、基础设施、配置和应用程序的底层操作系统 (OS)。所有这些更改都在单独的源代码存储库中进行版本控制。源代码依赖项（例如，库、编程语言和诸如 AMI ID 之类的参数）至少每周升级一次，自动升级到最新版本。

这些源使用与我们用于部署应用程序代码的相同安全机制（如自动回滚）部署到单独的管道中。 例如，在运行时可能会变化的服务的配置值（如 API 速率限制增加和功能标记）会自动部署在专用配置管道中。如果源更改导致服务在生产环境中出现任何问题（例如，未能解析配置文件），则源更改会自动回滚。

典型的微服务可能具有应用程序代码管道、基础设施管道、操作系统修补管道、配置/功能标记管道，以及运算符工具管道。 同一个微服务拥有多个管道有助于我们更快速地将更改部署到生产环境。 未通过集成测试且阻塞应用程序管道的应用程序代码更改不会影响其他管道。例如，它们不会阻止基础设施代码更改到达基础设施管道的生产阶段。 同一微服务的所有管道看起来都十分相似。例如，功能标记管道使用的安全部署技术与应用程序代码管道相同，因为错误的功能标记配置更改就像错误的应用程序代码更改一样，可能会影响生产。

代码审查

所有将要部署到生产环境的更改都从代码审查开始，并且必须得到团队成员的批准之后才能合并到*主线*分支（我们的“主要”或“主干”版本），主线分支会自动启动管道。 管道会强制要求所有合并到主线分支的提交内容都必须经过代码审查，并得到该管道服务团队成员的批准。管道将阻止部署任何未经过审查的提交内容。

对于完全自动化的管道，代码审查是代码更改在部署到生产环境之前最后一次接受工程师的人工审查和批准，所以这是关键步骤。 代码审查人员会对代码的正确性进行评估，还会评估是否可以将更改安全地部署到生产环境。 他们会评估代码是否经过充分的测试（单元测试、集成测试和 Canary 测试），是否充分执行以进行部署监控，以及是否可以安全地回滚。 有些团队使用像以下示例中所示的自定义检查清单，该检查清单会自动添加到每个团队的代码审查中，以明确检查是否存在部署安全性问题。

代码审查检查清单示例

```
## Testing
[ ] Did you write new unit tests for this change?
[ ] Did you write new integration tests for this change?
Include the test commands you ran locally to test this change:
```

```
mvn test && mvn verify
```

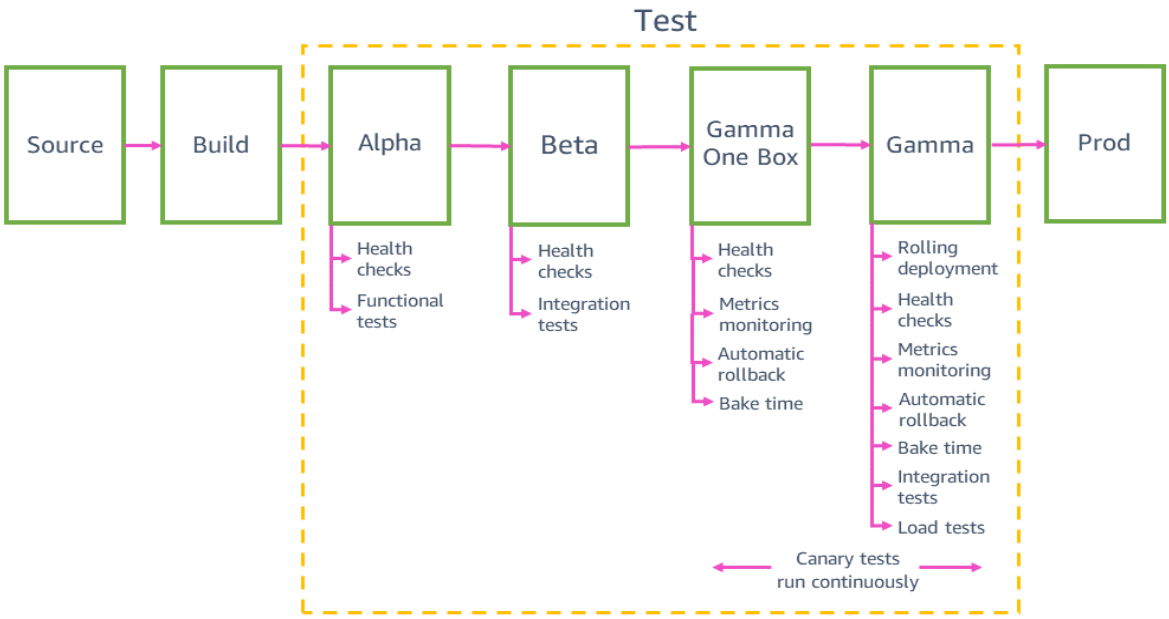

```
## Monitoring
[ ] Will this change be covered by our existing monitoring?
 (no new canaries/metrics/dashboards/alarms are required)
[ ] Will this change have no (or positive) effect on resources and/or
limits?
 (including CPU, memory, AWS resources, calls to other services)
[ ] Can this change be deployed to Prod without triggering any alarms?

## Rollout
[ ] Can this change be merged immediately into the pipeline upon approval?
[ ] Are all dependent changes already deployed to Prod?
[ ] Can this change be rolled back without any issues after deployment to
Prod?
```

构建测试和单元测试

代码在构建阶段进行编译和单元测试。构建工具和构建逻辑因不同的语言，甚至不同的团队而异。例如，团队可以选择最适合他们的单元测试框架、Linter 和静态分析工具。此外，团队还可以选择这些工具的配置，例如，在他们的单元测试框架中采用可接受的最低代码覆盖。 所运行的测试的工具和类型也因管道部署的代码类型而异。例如，单元测试用于应用程序代码，Linter 用于基础设施即代码模板。 所有内部版本在运行时都不连接网络以隔离内部版本，并支持内部版本的可再现性。通常情况下，单元测试会模仿（模拟）所有对依赖项的 API 调用，例如，其他 AWS 服务。与“实时”未模拟依赖项的集成稍后会在集成测试的管道中进行测试。 与集成测试相比，对模拟依赖项的单元测试能够对极端情况（例如，API 调用返回的意外错误）进行测试，并确保妥善处理代码中的错误。 构建完成后，编译后的代码会被打包并签名。

在预生产环境中测试部署

在部署到生产环境之前，管道会在多个预生产环境（例如，Alpha、Beta 和 Gamma）中部署并验证更改。 Alpha 和 Beta 通过运行功能 API 测试和端到端集成测试来验证最新代码是否按预期发挥功能。 Gamma 验证代码既能正常发挥功能，又可以安全地部署到生产环境。 Gamma 与生产环境极为相似，包括与生产环境相同的部署配置、相同的监控和警报，以及相同的持续 Canary 测试。 Gamma 还在多个 AWS 区域部署，以捕获由于区域差异而造成的任何潜在影响。



集成测试

集成测试可帮助我们自动使用服务，就像客户在管道中使用服务一样。 这些测试通过在所有有意义的客户场景的每个预生产阶段调用在真实基础设施上运行的真实 API，对整个堆栈进行端到端测试。集成测试的目的是，在部署到生产环境之前捕获服务的所有意外或错误行为。

尽管单元测试针对模拟的依赖项运行，但集成测试针对调用真实依赖项的预生产系统运行，验证关于这些依赖项的行为方式的模拟的假设。集成测试可验证单个 API 对不同输入的行为。此外，它们还验证加入了多个 API 的完整工作流，例如，创建新资源、描述新资源，直到资源准备就绪，然后使用这些资源。

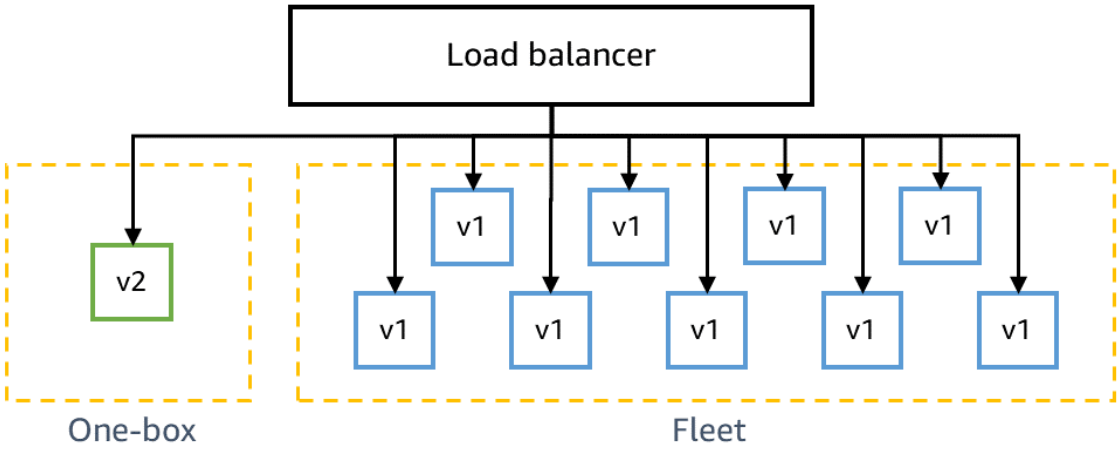
集成测试既运行正面测试案例，也运行负面测试案例，例如，为 API 提供无效输入，并检查是否按预期返回“输入无效”错误。 有些管道运行模糊测试以生成多个可能的 API 输入，并验证这些输入不会导致服务中出现任何内部故障。 有些管道还在预生产阶段运行短负载测试，以确保最新更改不会导致在实际负载级别出现任何延迟或吞吐量下降。

向后兼容性和 one-box 测试

在部署到生产环境之前，我们需要确保最新代码具有向后兼容性，并且能够安全地与当前代码一起部署。 例如，我们需要检测最新代码是否使用当前代码不能解析的格式写入数据。 Gamma 中的 *one-box* 阶段将最新代码部署到最小的部署单元，例如，部署到一个虚拟机或一个容器，或部署到一小部分 AWS Lambda 函数调用。 此 one-box 部署将 gamma 环境中的其余资源部署当前代码一段时间，例如，30 分钟或 1 小时。 流量不必专门驱动到一个盒中。可以添加到与剩余 Gamma 环境相同的负载均衡器，

或轮询相同的队列。例如，在负载均衡器后面包含 10 个容器的 Gamma 环境中，单盒会接收连续 Canary 测试生成的 10% 的 Gamma 流量。 one-box 部署会监控 Canary 测试成功率和服务指标，以检测由于部署或由于并排部署的“混合”队列所带来的任何影响。

下图显示了将新代码部署到 one-box 阶段但尚未部署到其他 gamma 队列后的 gamma 环境状态：



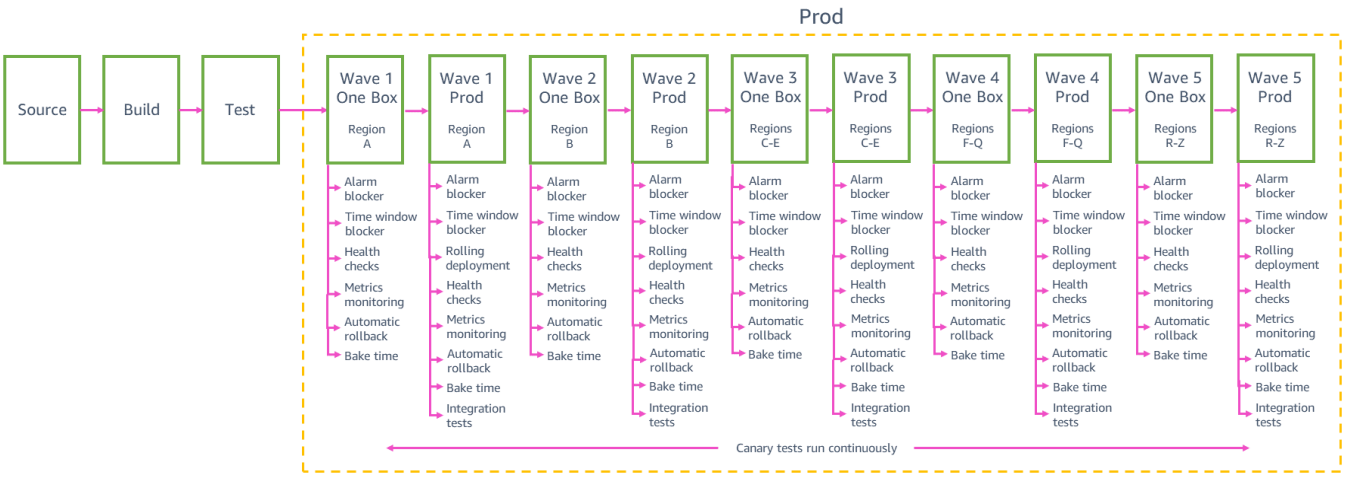
我们还需要确保最新代码可以与我们的依赖关系向后兼容，例如，是否需要以特定顺序在微服务间进行更改。 预生产环境中的微服务通常调用另一个团队拥有的任何服务的生产终端节点，例如 Amazon Simple Storage Service (S3) 或 Amazon DynamoDB，但它们调用微服务团队在同一阶段的其他微服务的预生产终端节点。 例如，一个团队在 gamma 中的微服务 A 调用同一团队在 gamma 中的微服务 B，但是它调用 Amazon S3 的生产终端节点。

有些管道还在一个称之为 *zeta* 的单独的向后兼容阶段再次运行集成测试。zeta 是一个单独的环境，在该环境中，每个微服务只调用生产终端节点，测试将要投产的更改是否与当前跨多个微服务在生产中部署的代码兼容。 例如，zeta 中的微服务 A 调用微服务 B 的生产终端节点和 Amazon S3 的生产终端节点。

有关如何编写和部署向后兼容更改的策略的描述，请参阅 Builders’ Library 文章确保部署期间的回滚安全。

生产部署

我们在 AWS 进行生产部署的首要目标是防止同时对多个区域以及同一区域的多个可用区造成负面影响。 限定每个单独部署的范围将会限定生产部署失败对客户的潜在影响范围，并防止对多可用区或多区域造成影响。 为了限定自动部署的范围，我们将管道的生产阶段分为多个阶段，并将多个部署拆分为单个区域。团队通过部署到管道中的单个可用区或服务的单个内部分区（称为*单元*），将区域部署拆分为范围更小的部署，以进一步限定生产部署失败的潜在影响范围。



交错部署

每个团队都需要权衡小范围部署的安全性与向所有区域的客户交付更改的速度之间的得失。 通过管道一次性将更改部署到 24 个区域或 76 个可用区，造成广泛影响的风险最低，但是管道可能需要数周时间才能向全球客户交付更改。 我们发现，如之前的示例生产管道所示，将部署分组成不断增大的“wave”（波段），有助于我们在部署风险和速度之间保持良好的平衡。 管道中的每个 wave 阶段都将部署编排到一组区域，更改从一个 wave 到另一个 wave，不断提升。 新更改可以随时进入到管道的生产阶段。在 wave 1 中，在将一组更改从第一步提升到第二步之后，下一组更改将从 gamma 被提升到 wave 1 的第一步，因此我们最终不会有大量的更改一直处于等待部署到生产环境中的状态。

管道中的前两个 wave 对于在更改中建立信心起到最关键的作用：第一个 wave 部署到请求数量少的区域，以限定新更改首次生产部署的潜在影响范围。 在该区域内，wave 一次仅部署到一个可用区（或单元），以谨慎地跨区域部署更改。然后，第二个 wave 将一次性部署到一个请求较多的区域中的一个可用区（或单元），客户很可能会在其中使用所有的新代码路径，我们也可以在此对更改进行良好的验证。

初步部署管道 wave 更改并对其安全性有了更高的信心之后，我们就可以在同一 wave 中并行部署到越来越多的区域中了。例如，之前的示例生产管道在 wave 3 中部署到了三个区域，在 wave 4 中部署到多达 12 个区域，然后在 wave 5 中部署到剩余区域。 每个 wave 中区域的准确数量和选择，以及服务团队管道中的 wave 数量，是取决于单个服务的使用模式和规模的。管道中后续的 wave 仍然有助于我们防止对同一区域中的多个可用区造成负面影响。当一个 wave 并行部署到多个区域时，它对每个区域都会遵循与初始 wave 相同的谨慎部署行为。wave 中的每个步骤都仅从 wave 中的每个区域部署到单个可用区或单元。

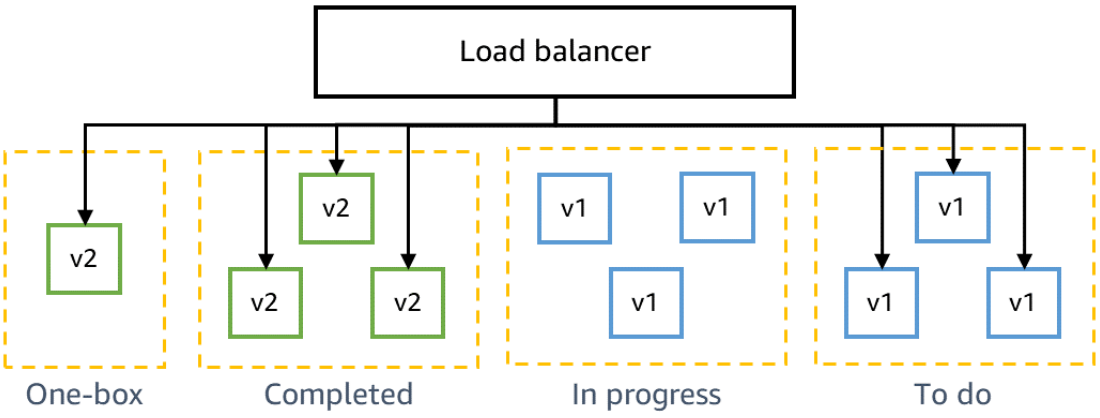
One-box 和滚动部署

每个生产 wave 的部署都是从 one-box 阶段开始。 与 gamma 的 one-box 阶段一样，每个生产 one-box 阶段都将最新代码部署到每个 wave 的区域或可用区中的一个 *box*（单个虚拟机、单个容器或一小部分 Lambda 函数调用）中。 生产 one-box 部署通过首先限制该 wave 中新代码所服务的请求，从而将更改对 wave 的潜在影响降到最小。 通常，一个 box 最多为区域或可用区总请求的 10% 提供服务。 如果更改在一个 box 中造成了负面影响，则管道会自动回滚更改，并且不会将其扩展到产品的其余阶段。

在 one-box 阶段之后，大多数团队使用滚动部署来部署到 wave 的主要生产队列。滚动部署可确保服务能够有足够的容量来在整个部署过程中为生产负载提供服务。它控制新代码*投入使用*的速度（即，当它开始为生产流量提供服务时），以降低更改的影响。通常，在区域滚动部署中，最多可将该区域中 33% 的服务 box（容器、Lambda 调用或虚拟机上运行的软件）替换为新代码。

在部署过程中，部署系统首先选择第一批至多 33% 的 box，来替换为新代码。更换期间，至少有 66% 的总容量是处于正常运行状态，可以为请求提供服务。 对所有服务进行缩放，以承受区域内失去一个可用区的情况，因此我们知道该服务仍可以以此容量为生产负载提供服务。 部署系统确定最第一批 box 中的一个 box 正在通过运行状况检查之后，剩余队列中的一个 box 可以用新代码替换，依此类推。同时，我们仍然始终保持至少 66% 的容量来为请求提供服务。 为了进一步限定更改的影响范围，部分团队的管道一次只部署了 5% 的设备。但是，它们随后执行*快速回滚*；在这一过程中，系统用以前的代码一次性替换 33% 的 box，从而加快回滚速度。

下图显示了滚动部署过程中生产环境的状态。 新代码已部署到 one-box 阶段，并已部署到第一批主要生产队列中。另一批已从负载均衡器中移除，并正在关闭以进行更换。



指标监控和自动回滚

通常情况下，在管道中的自动部署过程中，没有开发人员来主动监控每个部署的生产、检查指标以及在发现问题时手动回滚。完全无需对这些部署进行干预。部署系统会主动监控警报，以确定是否需要自动回滚部署。回滚会将环境切换回容器映像、AWS Lambda 函数部署程序包或之前已部署的内部部署程序包。 我们的内部部署程序包与容器映像类似，因为它们是不可变的，并使用校验和来验证其完整性。

每个区域中的每个微服务通常都具有一个严重级别较高的警报，它会触发影响服务客户的指标（如故障率和高延迟）阈值以及系统运行状况指标（如 CPU 利用率）阈值，如以下示例所示。 如果部署正在进行，则此严重级别较高的警报将用于呼叫工程师和自动回滚服务。 通常，在呼叫工程师并开始运行时，回滚已经在进行了。

严重级别较高的微服务警报示例

```
ALARM("FrontEndApiService_High_Fault_Rate") OR
ALARM("FrontEndApiService_High_P50_Latency") OR
ALARM("FrontEndApiService_High_P90_Latency") OR
ALARM("FrontEndApiService_High_P99_Latency") OR
ALARM("FrontEndApiService_High_Cpu_Usage") OR
ALARM("FrontEndApiService_High_Memory_Usage") OR
ALARM("FrontEndApiService_High_Disk_Usage") OR
ALARM("FrontEndApiService_High_Errors_In_Logs") OR
ALARM("FrontEndApiService_High_Failing_Health_Checks")
```

部署引入的更改可能会影响上游和下游微服务，因此部署系统需要监控正在部署的微服务的严重级别较高的警报，*并*监控团队其他微服务的严重级别较高的警报，以确定何时回滚。部署的更改也会影响连续的 canary 测试指标，因此部署系统还需要监控失败的 canary 测试。为了能够自动回滚所有这些可能受到影响的领域，团队会创建严重级别较高的聚合警报，供部署系统进行监控。严重级别较高的聚合警报将团队所有严重级别较高的单个微服务警报的状态和 Canary 警报的状态汇总为单个聚合状态，如以下示例所示。 如果团队微服务的任何严重级别较高的警报进入警报状态，则团队在该区域所有微服务中正在进行的所有部署将自动回滚。

严重级别较高的聚合回滚警报示例

```
ALARM("FrontEndApiService_High_Severity") OR
ALARM("BackendApiService_High_Severity") OR
ALARM("BackendWorkflows_High_Severity") OR
ALARM("Canaries_High_Severity")
```

One-box 阶段仅占总流量的一小部分，所以 one-box 部署所引入的问题可能不会触发该服务的严重级别较高的回滚警报。 为了及时捕获并回滚在 one-box 阶段导致问题但尚未到达其他生产阶段的变更，one-box 阶段还会回滚仅限定于这一个 box 的指标。 例如，它们会回滚专门由这一个 box 处理的请求的错误率，此类请求仅占请求总数的一小部分。

One-box 回滚警报示例

```
ALARM("High_Severity_Aggregate_Rollback_Alarm") OR
ALARM("FrontEndApiService_OneBox_High_Fault_Rate") OR
ALARM("FrontEndApiService_OneBox_High_P50_Latency") OR
ALARM("FrontEndApiService_OneBox_High_P90_Latency") OR
ALARM("FrontEndApiService_OneBox_High_P99_Latency") OR
ALARM("FrontEndApiService_OneBox_High_Cpu_Usage") OR
ALARM("FrontEndApiService_OneBox_High_Memory_Usage") OR
ALARM("FrontEndApiService_OneBox_High_Disk_Usage") OR
ALARM("FrontEndApiService_OneBox_High_Errors_In_Logs") OR
ALARM("FrontEndApiService_OneBox_Failing_Health_Checks")
```

除了回滚服务团队定义的警报以外，我们的部署系统还可以检测并自动回滚我们的内部 Web 服务框架发出的常见指标异常。 我们的大多数微服务以标准格式发出请求计数、请求延迟和故障计数等指标。利用这些标准指标，部署系统可以在部署过程中出现指标异常时自动回滚。此类情况包括请求计数突然下降到零，或者延迟或故障数量大大高于正常水平。

烘烤时间

由于部署而造成的负面影响有时不会立即显现出来。这就是*慢热的影响*。即：影响不会在部署过程中立即显现，尤其是当服务当时处于低负载状态时。 如果完成部署之后立即将变更推进到下一个管道阶段，那么当影响在第一个区域显现出来时，可能会有多个区域受到影响。 将变更推进到下一个生产阶段之前，管道中的每个生产阶段都设有*烘烤时间*。在此期间，管道在完成部署之后以及继续进行下一阶段之前继续监视团队的严重级别较高的聚合警报，以确认是否存在任何慢热的影响。

为了计算我们对部署进行烘烤所花费的时间，我们需要权衡两方面的风险：一方面是由于过快地将变更推进到多个区域而造成的更加广泛的影响，另一方面是我们向全球客户交付变更的速度。 我们发现有一种很好的方法可以平衡这些风险，就是在管道内较早的 wave 中留出较长的烘烤时间，等到我们对变更的安全性建立信心之后，较晚的 wave 就可以缩短烘烤时间。 我们的目标是最大限度地降低多个区域受到影响的风险。由于大多数部署不会得到团队成员的主动监视，典型管道的默认烘烤时间比较保守，而变更会在大约四五个工作日内部署到所有区域。在规模较大或至关重要的服务中，烘烤时间以及管道在全球范围内部署变更的时间更加保守。

典型的管道会在每个 one-box 阶段之后等待至少 1 小时，在第一个区域 wave 之后等待至少 12 小时，在其余区域 wave 之后等待至少 2 到 4 小时，并且会为单个区域、可用区以及每个 wave 中的单元留出额外的烘烤时间。 烘烤时间包括要求等待团队的指标达到特定的数据点数量（例如，“等待至少 100 个对 Create API 的请求”），从而确保发生了足够的请求，以便新代码得到充分执行。 在整个烘烤时间内，如果团队的严重级别较高的聚合警报进入警报状态，则部署将自动回滚。

尽管十分罕见，但在某些情况下，与管道对变更进行烘烤和部署所花费的时间相比，我们可能需要以更快的速度向客户交付紧急变更（例如安全修复或是对影响服务可用性的大型事件的缓解）。 在这种情况下，我们可以缩短管道的烘烤时间以加快部署速度，但是我们需要对为此进行的变更进行严格审查。对于此类情况，我们要求组织的首席工程师进行审查。 团队必须与经验丰富并在运行安全性方面拥有专长的开发人员合作，对代码变更及其紧迫性以及造成影响的风险进行审核。 变更仍然会在管道中遵循与平时相同的步骤，但是会更快地推进到下一阶段。 为了管理由于部署速度加快而造成的风险，我们对这段时间内管道中运行的变更采取限制，仅允许运行解决当前问题所需的最小代码变更；此外，我们还主动对部署进行监视。

警报和窗口拦截器

如果造成负面影响的风险更高，则管道会阻止自动部署到生产中。管道使用一组“拦截器”来评估部署风险。 例如，当环境中存在尚未解决的问题时，如果自动将新的变更部署到生产中，可能会造成更严重或更持久的影响。 在开始对任何生产阶段实施新部署之前，管道会检查团队的严重级别较高的聚合警报，以确定是否存在任何尚未解决的问题。 如果警报当前处于警报状态，则管道会阻止变更继续推进。 管道还可以检查整个组织范围内的警报，例如大型事件警报，该警报能够指示另一个团队的系统是否存在广泛影响，并阻止启动可能会造成更多整体性影响的新部署。 如果需要将变更部署到生产中，以便从严重级别较高的问题中恢复，开发人员可以覆盖这些部署拦截器。

此外，我们也为管道配置了一组时间窗口，它们定义了何时允许开始部署。 配置时间窗口时，我们需要权衡会造成部署风险的两方面原因。 一方面，如果时间窗口非常小，变更可能会在时间窗口关闭时堆积在管道中，从而增加这些变更在下一次部署中时间窗口开启时造成影响的可能性。另一方面，如果时间窗口非常大，超出了正常工作时间，由于部署失败而造成持久影响的风险就会增加。在下班时间，与待命工程师和其他团队成员白天一起工作时相比，与待命工程师取得联系需要花费更长时间。在正常工作时间内，如果在部署失败之后需要执行任何手动恢复步骤，可以更快地与团队取得联系。

大多数部署不会得到团队成员的主动监视，因此，在自动回滚之后需要手动操作的情况下，我们会优化部署时间安排，从而最大限度地缩短与待命工程师取得联系所需的时间。 在夜间、办公室假期以及周末，与待命工程师取得联系通常用时更久，所以这些时间不包括在时间窗口内。 视服务的使用模式而定，

某些问题可能在部署后几小时内不会显现出来，所以许多团队的时间窗口还排除了周五和下午较晚时间的部署，以降低部署后需要在夜间或周末与待命工程师取得联系的风险。 我们发现，即使在需要手动操作的情况下，这组时间窗口也能实现快速恢复，确保减少工作时间之外与待命工程师的联系，并确保在时间窗口关闭时将少量变更捆绑在一起。

管道即代码

典型的 AWS 服务团队拥有许多管道来部署团队的多种微服务和源类型（应用程序代码、基础设施代码、操作系统补丁等）。每个管道都针对数量不断增加的区域和可用区设有许多部署阶段。 这意味着团队需要进行大量配置，以便在管道系统、部署系统和警报系统中进行管理；此外，团队还需要付出大量努力来紧跟技术发展以实施最新的最佳实践，并及时了解新的区域和可用区。 过去几年中，我们积极实践“管道即代码”这种更具简便性和一致性的方法，以代码的形式对配置进行建模，由此配置安全的最新管道。 我们的内部管道即代码工具从区域和可用区域集中列表中拉取，可以轻松地将新的区域和可用区添加到整个 AWS 的管道中。 该工具还允许团队通过继承对管道进行建模，定义父类中团队管道的通用配置（例如进入每个 wave 的区域以及每个 wave 应该留出的烘烤时间长度），并将所有微服务管道配置定义为继承所有通用配置的子类。

结论

在 Amazon，我们以各种有助于平衡部署安全与部署速度的因素为基础，逐渐建立了我们的自动化部署安全实践。与此同时，我们希望最大限度地减少开发人员由于担心部署而花费的时间。 通过广泛的生产前测试、自动回滚以及交错生产部署，我们在发布流程中以自动化方式保证部署安全，从而最大限度地减少了部署可能对生产造成的影响。这意味着开发人员无需主动监视生产部署。

借助完全自动化的管道，开发人员可以通过代码审查来检查他们的代码，也可批准已准备好投入生产的变更。 将变更合并到源代码存储库中之后，开发人员可以继续执行下一个任务而不必为部署操心，因为值得信赖的管道能够以安全、谨慎的方式将变更投入生产。 这种自动化管道能够在一天之内持续多次部署到生产中，同时兼顾安全性和速度。 AWS 服务团队以代码的形式对持续交付实践进行建模，从而能够以前所未有的简便方式设置管道，由此安全地自动部署代码变更。