안전하고 간편한 배포 자동화

Clare Liguori

안전하고 간편한 배포 자동화

Amazon 입사 면접 당시 면접관 중 한 명에게 “얼마나 자주 프로덕션 환경에 배포합니까?”라는 질문을 했습니다. 당시 저는 메이저 릴리스가 연간 한두 차례 롤아웃되는 제품을 담당하고 있었는데, 그 사이에 사소한 수정 사항도 릴리스해야 했습니다. 수정 사항을 릴리스할 때마다 몇 시간씩 걸려 신중하게 롤아웃했습니다. 그리고는 배포 후에 잘못된 것이 없는지, 롤백해야 할지 살펴보기 위해 로그와 지표를 열심히 확인했습니다.

Amazon 이 지속적인 배포를 실행하고 있다는 내용을 읽은 적이 있어서 면접 때 Amazon 에서 개발자로서 배포를 관리하고 검증하는 데 얼마나 시간을 할애해야 할지 알고 싶었습니다. 면접관은 지속적 배포 파이프라인을 통해 변경 사항이 하루에 여러 번 프로덕션 환경에 자동으로 배포된다고 말했습니다. 제가 그랬듯이, 그런 배포를 각각 면밀하게 관리하고 영향이 있는지를 살피기 위해 로그와 지표를 모니터링하는 데 하루 중 몇 시간이나 할애하냐고 묻자 놀랍게도 전혀 그럴 필요가 없다고 답했습니다. 이 작업이 파이프라인에서 자동으로 이루어지고 대부분 배포의 경우 아무도 직접적으로 모니터링하는 사람은 없다는 것입니다. “우와!” 전 놀라서 소리쳤죠. Amazon 에 입사한 후 저는 이 ‘손 댈 필요 없는’ 자동화된 배포가 정확히 어떤 원리로 작동하는지 알아낼 생각에 신났습니다.

개발자의 작업 부담을 덜어주는 안전한 지속적 배포

그때부터 Amazon 이 빠르고 안전한 배포를 지원하기 위해 지속적 배포 파이프라인을 어떻게 구축하는지 직접 경험했습니다. 그리고 배포 작업에 들이는 시간으로부터 개발자들을 해방시켜준 안전한 지속적 배포 방식의 가치를 알게 되었습니다. 서비스 소스 코드 리포지토리의 메인 브랜치로 프로덕션 코드를 푸시하고 나면 더 이상의 작업이 필요 없고 다른 작업으로 넘어가면 됩니다. 팀의 파이프라인에서 해당 변경 사항을 프로덕션 환경에 자동으로 배포해주기 때문이죠. 코드 변경 사항을 프로덕션 서비스에 릴리스하는 프로세스는 파이프라인에 의해 완전 자동화되므로 소스 코드 리포지토리에 코드를 병합하고 난 후에는 저나 다른 개발자 누구도 코드를 다시 손보거나 검토할 필요가 없습니다.

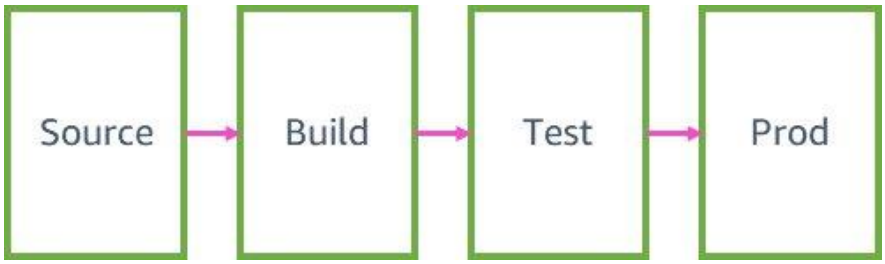
저희 팀은 변경 사항을 프로덕션 환경에 안전하게 배포하는 자동화된 단계로 파이프라인을 구축하기 때문에 배포를 매번 모니터링할 필요가 없습니다. 이 파이프라인은 최신 변경 사항에 대해 테스트와 배포 안전 검사를 실행합니다. 이 같은 자동화된 단계는 고객에게 영향을 미치는 결함이 프로덕션 환경에 유입되지 않도록 미연에 방지하고 프로덕션 환경에 유입된 경우에도 결함이 고객에게 미치는 영향을 최소화합니다. 개발자로서 저는 적극적으로 모니터링하지 않아도 이 파이프라인이 변경 사항을 프로덕션 환경에 안전하고 조심스럽게 배포해줄 것임을 신뢰할 수 있습니다.

지속적 전달이 실현되기까지

Amazon 이 처음부터 지속적 전달 방식을 사용한 것은 아니며, 개발자들이 코드를 프로덕션 환경에 배포하는 작업을 관리하는 데 몇 시간 또는 며칠을 할애하곤 했습니다. 소프트웨어를 배포하는 방식을 자동화/표준화하고 변경 사항을 프로덕션 환경에 적용하는 데 걸리는 시간을 줄일 대안으로서 전사적으로 지속적 전달 방식을 채용했습니다. 릴리스 프로세스는 장기간에 걸쳐 점진적으로 개선되었습니다. 배포와 관련한 위험을 파악하고 파이프라인에 새로운 안전 자동화 기능을 적용함으로써 그러한 위험을 완화할 방법을 찾았습니다. 그리고 새로운 위험과 배포의 안전을 보장할 새로운 방법을 계속 찾으면서 이 같은 릴리스 프로세스를 꾸준히 반복했습니다. 지속적 전달을 구현하기까지 과정과 지속적인 개선 방법에 대해 자세히 알아보려면 Builders' Library 문서 지속적 전달을 통한 신속한 배포를 참조하십시오.

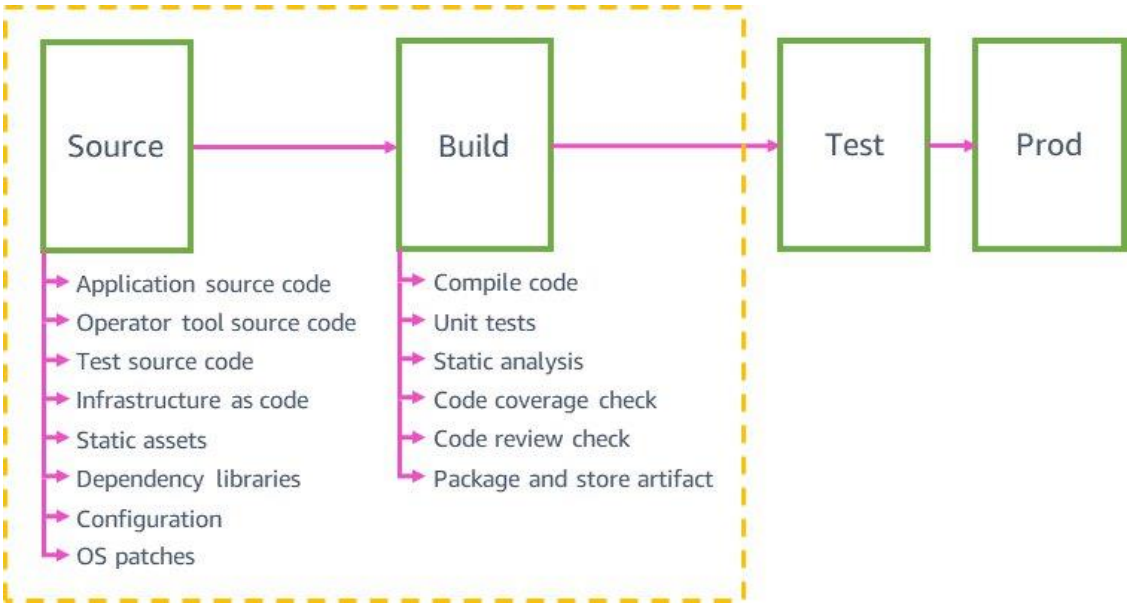
파이프라인의 4 단계

이 문서에서는 Amazon 에서 파이프라인을 통해 코드 변경 사항이 프로덕션 환경에 배포되는 과정을 단계별로 살펴봅니다. 일반적인 지속적 전달 파이프라인은 소스, 빌드, 테스트, 프로덕션이라는 4 개의 주요 단계로 구성됩니다. 일반적인 AWS 서비스의 각 파이프라인 단계를 자세히 설명보고 일반적인 AWS 서비스 팀이 파이프라인을 어떻게 설정하는지 보여주는 예제도 살펴보겠습니다.



소스 및 빌드

다음 다이어그램은 일반적인 AWS 서비스 팀 파이프라인의 소스 및 빌드 단계를 개략적으로 보여줍니다.



파이프라인 소스

Amazon 에서 파이프라인은 애플리케이션 코드 변경 사항만이 아니라 모든 유형의 소스 변경 사항을 검증하고 프로덕션 환경에 안전하게 배포합니다. 변경 사항을 검증하여 웹 사이트 정적 자산, 도구, 테스트, 인프라, 구성, 애플리케이션의 기반 OS(운영 체제)와 같은 소스에 배포할 수 있습니다. 모든 변경 사항은 개별 소스 코드 리포지토리에서 여러 버전으로 관리됩니다. 라이브러리, 프로그래밍 언어, 파라미터(예: AMI ID) 등의 소스 코드 종속성은 주 1 회 이상 최신 버전으로 자동 업그레이드됩니다.

이러한 소스는 애플리케이션 코드를 배포하는 데 사용하는 것과 동일한 안전 메커니즘(예: 자동 롤백)을 사용하여 개별 파이프라인에서 배포됩니다. 예를 들어 런타임에 변경될 수 있는 서비스의 구성 값(예: API 속도 제한 상승 및 기능 플래그)은 전용 구성 파이프라인에서 자동으로 배포됩니다. 소스 변경 사항이 프로덕션 환경에서 서비스에 문제를 유발할 경우(예: 구성 파일 구문 분석 오류) 자동으로 롤백됩니다.

일반적인 마이크로서비스에는 애플리케이션 코드 파이프라인, 인프라 파이프라인, OS 패치 적용 파이프라인, 구성/기능 플래그 파이프라인, 운영자 도구 파이프라인이 사용될 수 있습니다. 동일한 마이크로서비스에 여러 파이프라인을 사용함으로써 변경 사항을 프로덕션 환경에 보다 신속하게 배포할 수 있습니다. 통합 테스트를 통과하지 못하고 애플리케이션 파이프라인을 중단시키는 애플리케이션 코드 변경 사항이 있더라도 다른 파이프라인에는 영향을 미치지 않습니다. 예를 들어 인프라 파이프라인에서 인프라 코드 변경 사항이 프로덕션 환경에 적용되는 데에는 지장을 주지 않습니다. 동일한 마이크로서비스의 모든 파이프라인은 대체로 매우 유사합니다. 예를 들어 기능 플래그 파이프라인은

애플리케이션 코드 파이프라인과 같은 안전한 배포 기법을 사용합니다. 잘못된 기능 플래그 구성 변경 사항은 프로덕션 환경에서 잘못된 코드 변경 사항과 마찬가지로 악영향을 미칠 수 있기 때문입니다.

코드 검토

프로덕션 환경에 배포되는 모든 변경 사항은 코드 검토로 시작되며, 파이프라인을 자동으로 시작하는 메인라인 브랜치(AWS 가 사용하는 '마스터' 또는 '트렁크'를 일컫는 용어)에 병합하기 전에 팀원이 승인해야 합니다. 파이프라인에서는 메인라인 브랜치의 모든 커밋에 대해 해당 파이프라인의 서비스 팀원이 코드를 검토하고 승인하도록 요구합니다. 파이프라인은 검토되지 않은 커밋이 배포되지 않도록 차단합니다.

완전 자동화된 파이프라인에서 코드 검토는 엔지니어로부터 받은 코드 변경 사항을 프로덕션 환경에 배포하기 전에 마지막으로 이루어지는 수동 검토 및 승인 단계이므로, 매우 중요합니다. 코드 검토 담당자는 코드가 올바른지 확인하고 변경 사항을 프로덕션 환경에 안전하게 배포할 수 있는지도 평가합니다. 코드가 충분한 테스트(단위 테스트, 통합 테스트, Canary 테스트)를 거쳤는지, 배포 모니터링을 위한 도구는 충분한지, 안전하게 롤백할 수 있는지 등을 평가합니다. 일부 팀에서는 다음 샘플과 같은 맞춤형 체크리스트를 사용하기도 합니다. 이러한 체크리스트는 배포 안전성 문제를 명시적으로 점검하도록 팀의 코드 검토 건마다 자동으로 추가됩니다.

예제 코드 검토 체크리스트

```
## 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
```

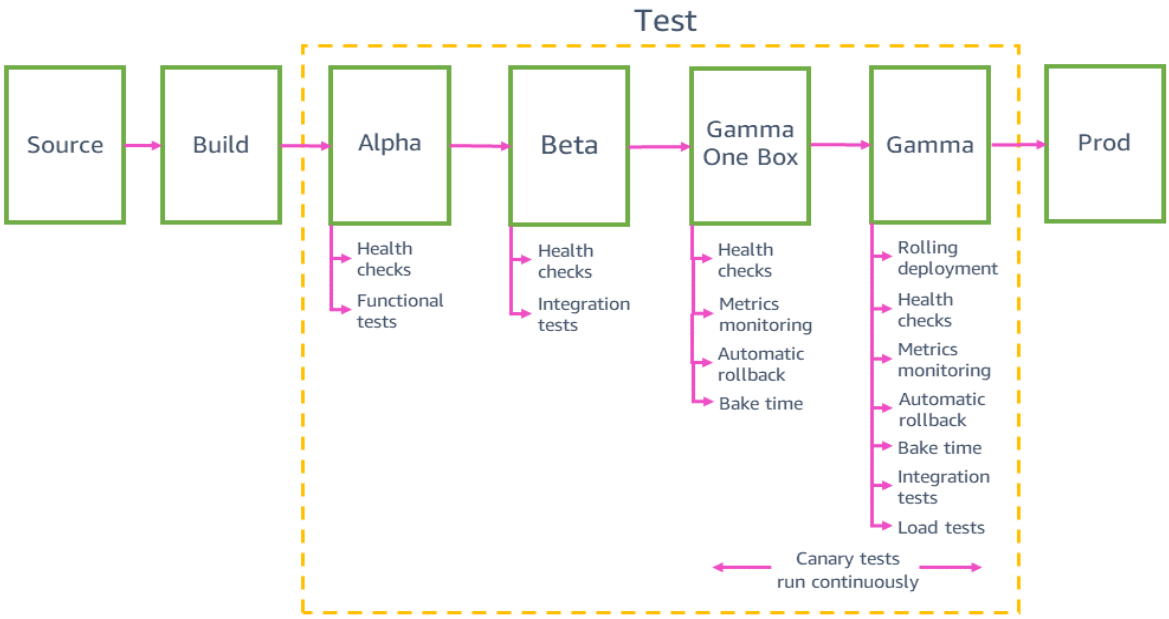
```
[ ] Can this change be merged immediately into the pipeline upon approval?
[ ] Are all dependent changes already deployed to Prod?
[ ] Can this change be rolled back without any issues after deployment to
Prod?
```

번들 및 단위 테스트

빌드 단계에서는 코드가 컴파일되고 단위 테스트를 거칩니다. 빌드 도구와 빌드 로직은 언어별로 다를 수 있고 팀마다 다를 수도 있습니다. 예를 들어 팀에서는 최적의 단위 테스트 프레임워크, Linter, 정적 분석 도구를 선택할 수 있습니다. 또한 팀에서는 단위 테스트 프레임워크의 최소 허용 코드 범위와 같은 도구 구성을 선택할 수도 있습니다. 실행하는 도구와 테스트 유형은 파이프라인에서 배포되는 코드의 유형에 따라 다릅니다. 예를 들어 단위 테스트는 애플리케이션 코드에 사용되고 Linter 는 코드형 인프라에 사용됩니다. 빌드를 격리하고 빌드 재현성을 높이기 위해 모든 빌드는 네트워크 액세스 없이 실행됩니다. 일반적으로 단위 테스트에서는 다른 AWS 서비스와 같은 종속성에 대한 모든 API 호출을 가상으로 실행(시뮬레이션)합니다. 가상이 아닌 '라이브' 종속성과의 상호 작용은 파이프라인에서 나중에 통합 테스트를 통해 테스트됩니다. 통합 테스트와 달리, 가상의 종속성을 사용하는 단위 테스트는 API 호출에서 반환되는 예상치 못한 오류와 같은 엣지 사례를 실행하여 코드에서 오류가 정상적으로 처리되도록 보장할 수 있습니다. 빌드가 완료되면 컴파일된 코드를 패키징하고 서명합니다.

프로덕션 전 환경에서의 테스트 배포

파이프라인에서는 프로덕션 환경에 배포하기 전에 변경 사항을 알파, 베타, 감마 등 여러 프로덕션 전 환경에 배포하여 검증합니다. 알파 및 베타에서는 기능적인 API 테스트와 엔드 투 엔드 통합 테스트를 실행하여 최신 코드가 예상대로 작동하는지 검증합니다. 감마에서는 코드가 정상적으로 작동하고 프로덕션 환경에 안전하게 배포할 수 있는지 검증합니다. 감마는 프로덕션 환경과 동일한 배포 구성, 동일한 모니터링 및 경보 기능, 동일한 지속적 Canary 테스트 등을 사용하여 최대한 프로덕션 환경과 가깝게 구성됩니다. 또한 리전별 차이로 인해 발생할 수 있는 잠재적 영향을 포착하기 위해 감마는 여러 AWS 리전에 배포됩니다.



통합 테스트

통합 테스트는 파이프라인의 일부로서 고객이 사용하는 것과 같은 방식으로 서비스를 자동으로 사용해볼 수 있게 합니다. 이 같은 테스트는 각 프로덕션 전 단계에서 유의미한 고객 시나리오를 상정하여 실제 인프라에서 실행되는 실제 API 를 호출함으로써 전체 스택을 엔드 투 엔드로 실행합니다. 통합 테스트의 목적은 프로덕션 환경에 배포하기 전에 서비스의 예상치 못한 동작이나 잘못된 동작을 찾아내는 것입니다.

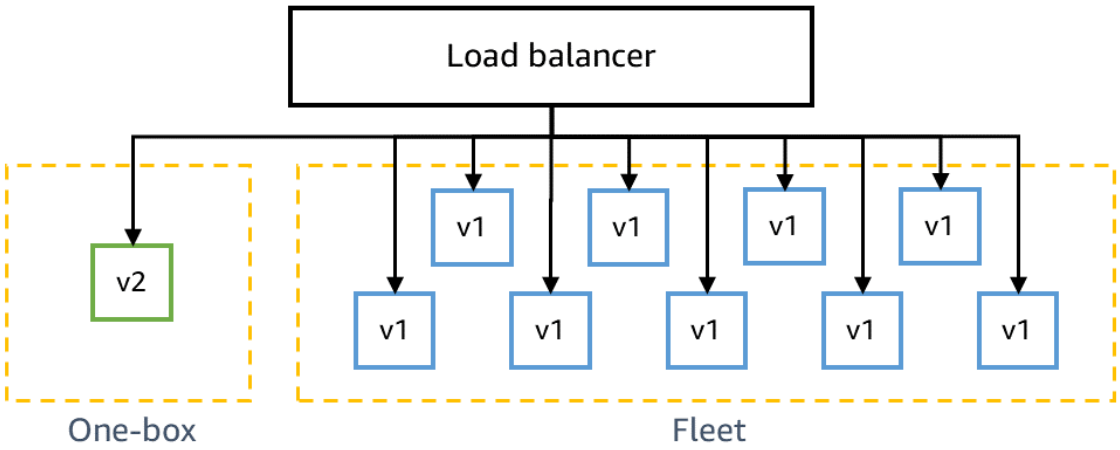
통합 테스트에서는 가상의 종속성에 대해 단위 테스트를 실행하는 동시에, 프로덕션 전 시스템에 대해 실제 종속성을 호출하는 테스트를 실행함으로써 그러한 종속성의 작동 방식에 대한 가설을 검증합니다. 통합 테스트는 다양한 입력별로 개별 API 의 동작을 검증합니다. 또한 새 리소스를 생성하고, 준비될 때까지 새 리소스를 설명한 다음, 리소스를 사용하는 여러 API 를 연결하는 전체 워크플로도 검증합니다.

통합 테스트에서는 긍정적 테스트 사례는 물론, API 에 잘못된 입력을 제공하여 '잘못된 입력' 오류가 정상적으로 반환되는지 확인하는 등의 부정적 테스트 사례도 실행합니다. 일부 파이프라인은 여러 가지 가능한 API 입력을 생성하여 서비스에서 내부 오류를 유발하지 않는지 검증하는 퍼즈(fuzz) 테스트도 실행합니다. 일부 파이프라인은 프로덕션 전 단계에서 최신 변경 사항이 실제 로드 수준에서 지연 시간 또는 처리량 저하를 유발하지 않는지 확인하는 단기 로드 테스트도 실행합니다.

이전 버전과의 호환성 및 원박스(One-box) 테스트

프로덕션 환경에 배포하기 전에 최신 코드가 이전 버전과 호환되고 현재 코드와 함께 배포해도 안전한지 확인해야 합니다. 예를 들어 최신 코드가 현재 코드에서 구문 분석할 수 없는 형식으로 코드를 작성할 경우 이를 감지해내야 합니다. 감마의 원박스 단계에서는 단일 가상 머신이나 단일 컨테이너, 작은 비율의 AWS Lambda 함수 호출과 같은 최소 배포 단위에 최신 코드를 배포합니다. 이 원박스 배포에서 나머지 감마 환경은 일정 시간(예: 30 분 또는 1 시간) 동안 현재 코드가 배포된 상태로 유지됩니다. 트래픽을 특별히 원박스로 유도할 필요는 없습니다. 나머지 감마 환경과 동일한 로드 밸런서에 추가하거나 동일한 대기열을 폴링하면 됩니다. 예를 들어 10 개의 컨테이너가 로드 밸런서를 통해 연결된 감마 환경에서 원박스는 지속적인 Canary 테스트에서 생성되는 감마 트래픽의 10%를 수신합니다. 원박스 배포에서는 Canary 테스트 성공률과 서비스 지표를 모니터링하여 배포에 따른 영향 또는 '혼합' 플릿을 나란히 배포하는 데 따른 영향을 감지합니다.

다음 다이어그램은 새 코드가 원박스 단계에 배포되고 감마 플릿의 나머지 부분에는 아직 배포되지 않은 감마 환경의 상태를 보여줍니다.



특정 순서로 여러 마이크로서비스를 변경해야 하는지 등, 최신 코드가 기존 종속성과 호환되는지 확인해야 합니다. 프로덕션 전 환경의 마이크로서비스는 일반적으로 Amazon Simple Storage Service(S3), Amazon DynamoDB 등 다른 팀이 소유한 서비스의 프로덕션 엔드포인트를 호출하지만, 같은 단계에 있는 해당 서비스 팀의 다른 마이크로서비스에 대해서는 프로덕션 전 엔드포인트를 호출하게 됩니다. 예를 들어 팀의 감마 단계인 마이크로서비스 A 는 동일한 팀의 감마 단계인 마이크로서비스 B 를 호출하지만 Amazon S3 에 대해서는 프로덕션 엔드포인트를 호출합니다.

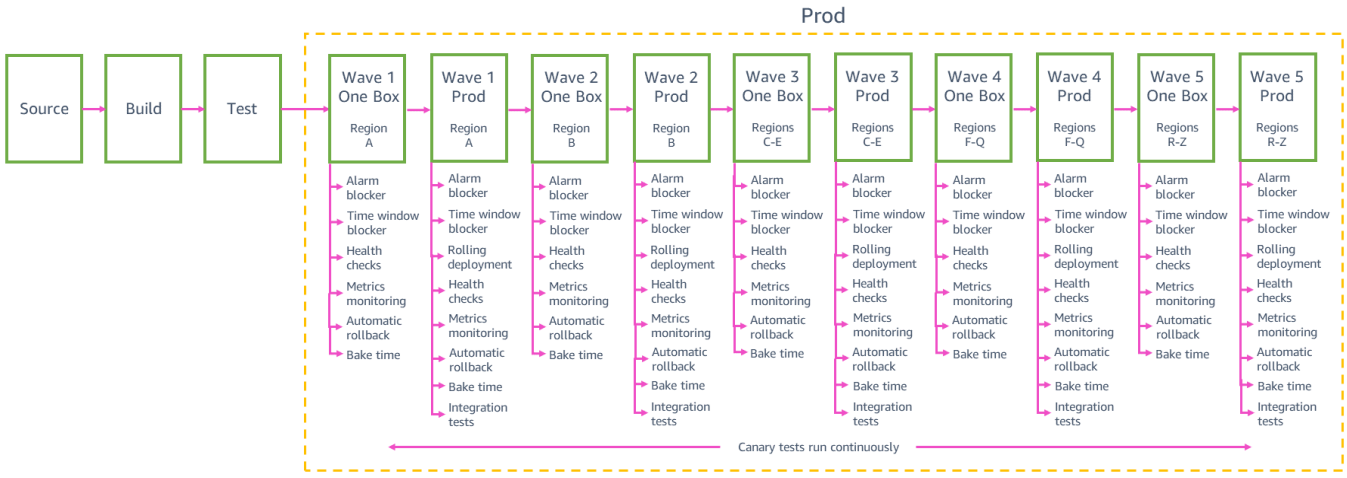
일부 파이프라인은 제타라고 하는 별도의 이전 버전과의 호환성 단계에서 통합 테스트를 다시 실행하기도 합니다. 제타는 각 마이크로서비스가 프로덕션 엔드포인트만 호출하여 프로덕션 환경에 배포될 변경

사항이 여러 마이크로서비스에 걸쳐 현재 프로덕션 환경에 배포된 코드와 호환되는지 테스트합니다. 예를 들어 제타 환경에서 마이크로서비스 A 는 마이크로서비스 B 의 프로덕션 엔드포인트와 Amazon S3 의 프로덕션 엔드포인트를 호출합니다.

이전 버전과 호환되는 변경 사항을 작성하고 배포하기 위한 전략에 대한 서명은 Builders' Library 문서 배포 중 롤백 안전 보장을 참조하십시오.

프로덕션 배포

AWS 에서 프로덕션 배포의 가장 중요한 목표는 여러 리전 및 동일한 리전 내 여러 가용 영역에 동시에 부정적 영향을 미치지 않도록 하는 것입니다. 개별 배포의 범위를 한정함으로써 실패한 프로덕션 배포가 고객에게 미치는 잠재적 영향을 제한하고 여러 가용 영역 또는 여러 리전에 걸쳐 영향을 미치지 않도록 방지할 수 있습니다. 자동 배포의 범위를 제한하기 위해 파이프라인의 프로덕션 배포를 여러 단계로 나누고 개별 리전에도 여러 번 나누어 배포합니다. 팀에서는 개별 가용 영역에 배포하거나 해당 서비스의 개별 내부 샤드(셀이라고 함)에 배포하는 방식으로 파이프라인에서 리전별 배포를 더 작은 범위의 배포로 분할하여, 실패한 프로덕션 배포가 미치는 잠재적 영향의 범위를 더욱 제한합니다.



단계별 배포

각 팀은 이러한 작은 범위의 배포에서 안전성과 속도의 균형을 확보하여 모든 리전의 고객에게 변경 사항을 제공할 수 있도록 해야 합니다. 파이프라인을 통해 한 번에 한군데씩 24 개 리전 또는 76 개 가용 영역에 차례로 변경 사항을 배포하면 광범위한 영향을 미칠 위험이 가장 적지만, 파이프라인에서 전 세계 고객에게 변경 사항을 제공하는 데 몇 주가 걸릴 수 있습니다. 전 샘플 프로덕션 파이프라인에서 보듯이 배포를 점차로 크기가 커지는 '웨이브'라는 그룹으로 분류하면 배포 위험과 속도의 적절한 균형을 유지할 수

있습니다. 파이프라인에서 각 웨이브의 단계는 배포를 리전의 특정 그룹으로 오케스트레이션하며, 변경 사항은 이러한 웨이브에서 다음 웨이브로 승격됩니다. 새로운 변경 사항은 언제든 파이프라인의 프로덕션 단계에 진입할 수 있습니다. 웨이브 1 의 변경 사항 세트가 첫 단계에서 두 번째 단계로 승격되면, 감마의 다음 변경 사항 세트가 웨이브 1 의 첫 단계로 승격되는 식이므로 프로덕션에 배포할 방대한 양의 변경 사항이 대기하는 일은 없습니다.

파이프라인의 처음 2 개 웨이브에서 변경 사항의 신뢰성이 가장 크게 확보됩니다. 첫 번째 웨이브에서는 새로운 변경 사항의 첫 번째 프로덕션 배포가 미칠 수 있는 영향을 제한하기 위해 요청 건수가 적은 리전에 배포합니다. 이 웨이브는 해당 리전에서 한 번에 하나의 가용 영역(또는 셀)에만 배포하면서 변경 사항을 신중하게 리전 전체에 배포합니다. 다음으로 두 번째 웨이브에서는 요청 건수가 많은 리전에서 한 번에 하나의 가용 영역(또는 셀)에 배포합니다. 이러한 리전은 고객이 모든 새로운 코드 경로를 실행할 가능성이 높으므로 변경 사항을 확실히 검증할 수 있습니다.

초기 파이프라인 웨이브의 배포에서 변경 사항의 안전성에 대한 신뢰도가 높아지면 점차로 리전 수를 늘리면서 동일한 웨이브에서 동시에 여러 리전에 배포할 수 있습니다. 예를 들어 이전 샘플 프로덕션 파이프라인은 웨이브 3 에서 3 개 리전에 배포한 후, 웨이브 4 에서는 최대 12 개 리전에, 웨이브 5 에서는 나머지 리전에 각각 배포합니다. 각 웨이브의 정확한 리전 수와 리전 선택 방식, 그리고 서비스 팀 파이프라인의 웨이브 수는 개별 서비스의 사용 패턴과 규모에 따라 달라집니다. 파이프라인 후반부의 웨이브는 동일한 리전의 여러 가용 영역에 미치는 부정적인 영향을 방지한다는 목표를 실현하는 데 도움이 됩니다. 웨이브에서 여러 리전에 동시에 배포할 때에도 초기 웨이브에서 각 리전에 배포할 때 사용된 것과 같은 신중한 롤아웃 동작을 따릅니다. 웨이브의 각 단계는 해당 웨이브의 각 리전에서 단일 가용 영역 또는 셀에만 배포합니다.

원박스(One-box) 및 롤링 배포

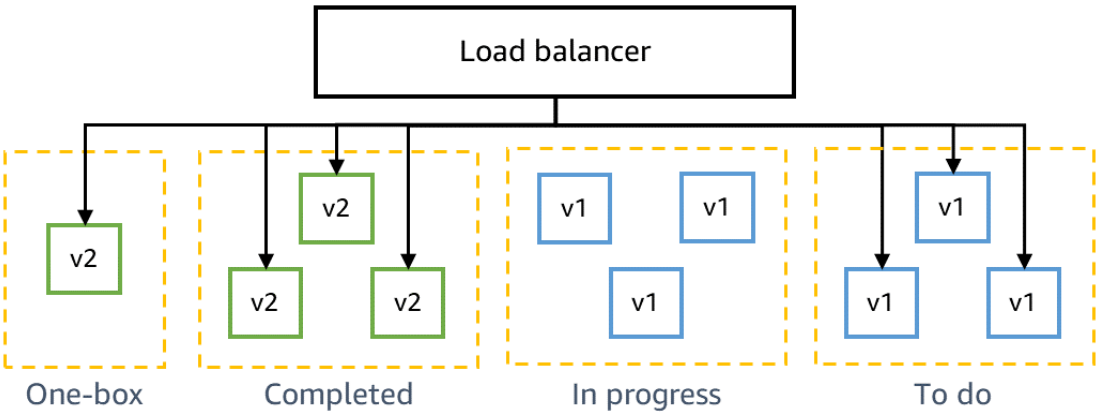
각 프로덕션 웨이브로의 배포는 원박스 단계부터 시작됩니다. 감마 원박스 단계와 마찬가지로, 각 프로덕션 원박스 단계에서는 해당 웨이브의 각 리전 또는 가용 영역에 있는 원박스(단일 가상 머신, 단일 컨테이너 또는 Lambda 함수 호출의 작은 비율)에 최신 코드를 배포합니다. 프로덕션 원박스 배포는 해당 웨이브에서 새로운 코드로 처리되는 요청의 수를 초기에 제한하는 방법으로 변경 사항이 웨이브에 미치는 잠재적 영향을 최소화합니다. 일반적으로 원박스는 리전 또는 가용 영역의 전체 요청 중 10% 미만을 처리합니다. 변경 사항이 원박스에서 부정적이 영향을 유발할 경우 파이프라인은 변경 사항을 자동으로 롤백하고 나머지 프로덕션 단계로 승격하지 않습니다.

원박스 단계를 거친 후에는 대부분의 팀이 롤링 배포를 통해 웨이브의 주 프로덕션 플릿에 배포합니다. 롤링 배포는 서비스가 전체 배포 시에 프로덕션 로드를 처리하기에 충분한 용량을 갖추도록 보장합니다.

새로운 코드를 서비스에 적용하는 속도(프로덕션 트래픽을 처리하기 시작하는 시점)를 통제하여 변경 사항의 영향을 제한합니다. 특정 리전에 대한 일반적인 롤링 배포에서는 해당 리전에서 제공되는 서비스의 박스(컨테이너, Lambda 호출 또는 가상 머신에서 실행되는 소프트웨어) 중 최대 33%가 새로운 코드로 교체됩니다.

배포 과정에서 배포 시스템은 먼저 새 코드로 교체할 최대 33%의 초기 박스군을 선택합니다. 교체 중에는 전체 용량 중 최소 66%가 정상 상태로 요청을 처리합니다. 모든 서비스는 리전에서 가용 영역 중 하나에 장애가 발생하더라도 정상 작동하도록 설계되므로, 이 용량으로 충분히 프로덕션 로드를 처리할 수 있습니다. 배포 시스템이 초기 박스군의 특정 박스가 상태 확인을 통과한 것을 확인하면, 나머지 플릿의 박스를 새 코드로 교체하는 식으로 진행됩니다. 그 동안에도 항상 용량의 최소 66%가 정상 상태로 유지되면서 요청을 처리합니다. 변경 사항이 미치는 영향을 더욱 제한하기 위해 일부 팀의 파이프라인에서는 한 번에 단 5%의 박스에만 배포합니다. 하지만 이 경우 신속한 롤백을 통해 한 번에 33%의 박스를 이전 코드로 교체하여 롤백 속도를 높입니다.

다음 다이어그램은 롤링 배포 중인 프로덕션 환경의 상태를 보여줍니다. 새 코드가 원박스 단계에서 주 프로덕션 플릿의 첫 박스군에 배포되었습니다. 다른 박스군은 로드 밸런서에서 제거되고 교체를 위해 종료하는 중입니다.



지표 모니터링 및 자동 롤백

파이프라인의 자동화된 배포에서는 일반적으로 프로덕션 환경으로의 각 배포 건을 개발자가 적극적으로 모니터링하거나, 지표를 점검하거나, 문제가 발견될 경우 수동으로 롤백하지 않습니다. 이러한 배포는 완전히 자동으로 실행됩니다. 배포 시스템은 경보를 적극적으로 모니터링하여 배포를 자동으로 롤백해야 할지 결정합니다. 롤백은 환경을 이전에 배포된 컨테이너 이미지, AWS Lambda 함수 배포 패키지 또는

내부 배포 패키지로 되돌립니다. 내부 배포 패키지는 변경이 불가능하고 체크섬을 사용하여 무결성을 확인하다는 점에서 컨테이너 이미지와 유사합니다.

각 리전의 각 마이크로서비스에는 일반적으로 높은 심각도 경보가 구성되어 있습니다. 다음 예제에서 보듯이 이 경보는 서비스의 고객에게 영향을 미치는 지표(예: 오류 발생률 및 긴 지연 시간)의 임계값에 도달할 경우, 그리고 시스템 상태 지표(예: CPU 사용률)에 따라 트리거됩니다. 높은 심각도 경보는 대기 엔지니어를 호출하고 배포가 진행 중인 경우 롤백하는 데 사용됩니다. 대기 엔지니어가 호출되어 대응을 시작할 때는 이미 롤백이 진행 중인 경우가 많습니다.

높은 심각도 마이크로서비스 경보 예제

```
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")
```

배포를 통해 적용되는 변경 사항은 업스트림 및 다운스트림 마이크로서비스에 영향을 미치므로 배포 시스템은 배포 중인 마이크로서비스에 대한 높은 심각도 경보를 모니터링하고 팀의 다른 마이크로서비스에 대한 높은 심각도 경보를 모니터링하여 언제 롤백할지 결정해야 합니다. 배포된 변경 사항은 지속적 Canary 테스트의 지표에도 영향을 미치므로 배포 시스템은 실패한 Canary 테스트도 모니터링해야 합니다. 영향을 미칠 수 있는 이러한 모든 영역에서 자동으로 롤백하기 위해 팀에서는 배포 시스템이 모니터링할 높은 심각도 집계 경보를 생성합니다. 다음 샘플과 같이, 높은 심각도 집계 경보는 팀의 모든 개별 마이크로서비스 높은 심각도 경보의 상태와 Canary 경보의 상태를 단일 집계 상태로 롤업합니다. 팀의 마이크로서비스에 대한 높은 심각도 경보 중 하나라도 경보 상태가 되면 팀의 모든 마이크로서비스에 걸쳐 해당 리전에서 진행 중인 모든 배포가 자동으로 롤백됩니다.

높은 심각도 집계 롤백 경보 예제

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

원박스 단계에서는 전체 트래픽의 일부만 처리하므로 원박스 배포로 인해 발생하는 문제는 서비스의 높은 심각도 롤백 경보를 트리거하지 않습니다. 원박스 단계에서 발생하는 문제가 나머지 프로덕션 단계에 도달하기 전에 포착하여 변경 사항을 롤백하기 위해 원박스 단계에서는 원박스만 대상으로 하는 지표를 기준으로도 추가로 롤백합니다. 예를 들어 전체 요청 건수에서 극히 일부만 차지하는 원박스에서 처리되는 요청의 오류 발생률이 높을 경우 롤백합니다.

원박스 롤백 경보 예제

```
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")
```

서비스 팀에서 정의한 경보를 기준으로 롤백하는 것 외에, 배포 시스템은 내부 웹 서비스 프레임워크에서 생성되는 공통 지표의 이상치를 기준으로도 자동으로 롤백합니다. 대부분의 마이크로서비스는 요청 건수, 요청 지연 시간, 오류 건수와 같은 지표를 표준 형식으로 생성합니다. 배포 시스템은 이러한 표준 지표를 사용함으로써 배포 중에 지표에서 이상치가 나타나면 자동으로 롤백할 수 있습니다. 요청 건수가 갑자기 0 으로 떨어지거나 지연 시간 또는 장애 건수가 평소보다 높아지는 경우를 예로 들 수 있습니다.

베이크 시간

배포로 인한 부정적인 영향이 바로 나타나지 않는 경우도 있습니다. 이를 슬로우 버닝이라고 합니다. 특히 배포 당시 서비스의 로드가 적을 경우 배포 시에 즉각적으로 문제가 나타나지 않는 형상을 말합니다. 이 경우 배포가 완료된 후 변경 사항을 다음 파이프라인 단계로 승격하면 첫 번째 리전에서 영향이 나타날 때에는 이미 여러 리전에서 영향을 미치고 있을 수 있습니다. 파이프라인의 각 프로덕션 단계에서는 변경 사항을 다음 프로덕션 단계로 승격하기 전에 베이크 시간을 적용합니다. 배포가 완료되고 다음 단계로 넘어가기 전에 이 시간 동안 파이프라인은 팀의 높은 심각도 집계 경보를 지속적으로 모니터링하여 슬로우 버닝 현상이 나타나지 않는지 확인합니다.

배포의 베이크에 할애할 시간을 계산하려면 변경 사항을 너무 빨리 여러 리전으로 승격함으로써 영향을 미치는 범위가 넓어질 위험과 변경 사항을 전 세계 고객에게 제공하는 속도 간에 적절한 균형점을 찾아야

합니다. 파이프라인의 초기 웨이브에서 이러한 위험성의 균형점을 찾는 효과적인 방법은 변경 사항의 안전성에 대한 신뢰성을 확보하는 동안에는 베이크 시간을 늘리고 이후 웨이브에서는 베이크 시간을 줄이는 것입니다. 여러 리전에 영향을 미칠 위험을 최소화하는 것이 목표입니다. 대부분의 배포는 팀원이 적극적으로 모니터링하지 않으므로 일반적인 파이프라인의 기본 베이크 시간은 보수적으로 설정되며 영업일을 기준으로 4~5 일 내에 모든 리전으로 변경 사항이 배포됩니다. 규모가 더 크거나 중요도가 높은 서비스의 경우 베이크 시간과 파이프라인에서 변경 사항을 전 세계적으로 배포하는 시간을 더 보수적으로 설정할 수 있습니다.

일반적인 파이프라인은 각 원박스 단계 이후 최소 1 시간, 첫 번째 리전별 웨이브 후에는 12 시간 이상, 나머지 각각의 리전별 웨이브 이후에는 2~4 시간 이상 대기하며, 각 웨이브 내의 개별 리전, 가용 영역 및 셀에 대해 추가 베이크 시간을 적용합니다. 베이크 시간에는 팀의 지표에 일정 개수의 데이터 포인트가 수집될 때까지 기다리는 요구 사항도 포함됩니다(예: 'Create API 에 대한 요청이 100 건 이상 발생할 때까지 대기'). 이는 새 코드가 완벽하게 실행될 정도로 충분한 요청이 발생할 때까지 기다리기 위한 동작입니다. 전체 베이크 시간 동안 팀의 높은 심각도 집계 경보가 경보 상태로 전환될 경우 배포가 자동으로 롤백됩니다.

매우 드물기는 하지만, 파이프라인이 통상적으로 변경 사항을 베이크하고 배포하는 데 걸리는 시간보다 빠르게 긴급한 변경 사항(예: 보안 수정 사항 또는 서비스 가용성에 영향을 미치는 대규모 이벤트에 대한 완화 조치)을 고객에게 제공해야 할 경우도 있습니다. 이 경우 파이프라인의 베이크 시간을 줄여 배포 속도를 높일 수 있지만, 이를 위해서는 변경 사항에 대한 면밀한 조사가 필요합니다. 이 경우 조직의 수석 엔지니어가 조사를 실시해야 합니다. 팀의 숙련된 개발자이자 운영 안전성 전문가가 코드 변경 사항과 긴급성 및 영향을 미칠 위험성을 검토해야 합니다. 이 경우에도 변경 사항은 정상적으로 파이프라인의 모든 단계를 거치지만 좀 더 빠르게 다음 단계로 승격됩니다. 이 시간 동안에는 파이프라인에서 실행 중에 변경 사항을 제한하는 방식으로 현재 문제를 해결하는 데 필요한 최소한의 변경 사항만 허용하고, 배포를 적극적으로 모니터링함으로써 빠른 배포의 위험성을 관리합니다.

경보 및 시간대 차단기

파이프라인에서는 부정적인 영향을 미칠 위험이 높은 시간대에 프로덕션 환경으로의 자동 배포를 방지합니다. 파이프라인은 배포 위험성을 평가하는 일련의 '차단기'를 사용합니다. 예를 들어 현재 환경에서 문제가 지속되고 있는데 새 변경 사항을 프로덕션 환경에 자동으로 배포하면 부정적인 영향이 더 커지거나 더 오래 지속될 수 있습니다. 프로덕션 단계로의 새 배포를 시작하기 전에 파이프라인은 팀의 높은 심각도 집계 경보를 검사하여 진행 중인 문제가 없는지 확인합니다. 경보가 현재 경보 발생 상태인 경우 파이프라인이 변경 사항의 적용을 방지합니다. 또한 파이프라인은 다른 팀의 시스템에 미치는 영향이 있는지를 나타내는 대규모 이벤트 경보와 같은 전사적인 경보를 확인하고 전반적인 영향을 가중시킬 수

있는 새 배포의 시작을 방지할 수도 있습니다. 심각도가 높은 문제로부터 복구하기 위해 프로덕션 환경에 변경 사항을 배포해야 할 경우 개발자가 이러한 배포 차단기를 무시할 수 있습니다.

또한 파이프라인에는 배포를 시작하도록 허용되는 시간을 정의하는 일련의 시간대도 구성되어 있습니다. 시간대를 구성할 때는 배포의 위험을 초래하는 두 가지 원인을 균형 있게 고려해야 합니다. 우선, 시간대가 매우 짧으면 나머지 시간 동안 파이프라인에 변경 사항이 누적되어 다음에 해당 시간대가 도래했을 때 배포되는 변경 사항 중에 영향을 미치는 변경 사항이 포함될 확률이 높아집니다. 반면 시간대가 통상적인 근무 시간을 넘길 정도로 너무 길면 실패한 배포의 영향이 장시간 이어질 위험이 높아집니다. 근무 시간 외에는 대기 엔지니어와 다른 팀원들이 모두 일하고 있는 근무 시간 때보다 대기 중인 엔지니어가 문제에 대응하는 데 더 오래 걸립니다. 통상적인 근무 시간 중에는 실패한 배포 후 수동 복구 조치가 필요할 때 팀이 더 빠르게 대응할 수 있습니다.

대부분의 배포는 팀원이 적극적으로 모니터링하지 않으므로, 자동 롤백 후에 복구하기 위해 수동 조치가 필요한 경우 대기 엔지니어가 대응하는 데 걸리는 시간이 최소화되도록 배포 타이밍을 최적화해야 합니다. 일반적으로 야간, 회사 휴무일 및 주말에는 대기 엔지니어가 대응하는 데 더 오래 걸리므로 이러한 시간은 배포 시간대에서 제외됩니다. 서비스의 사용 패턴에 따라 일부 문제는 배포 후 몇 시간 동안 드러나지 않을 수 있으므로, 배포 후 야간이나 주말에 대기 엔지니어가 대응해야 할 가능성을 줄이기 위해 금요일과 오후 늦은 시간도 배포 시간대에서 제외하는 팀이 많습니다. 이 같은 시간대 설정은 수동 조치가 필요한 경우에도 빠른 복구를 가능하게 하고, 대기 엔지니어가 통상적인 근무 시간 외에 대응해야 하는 사례를 줄이며, 나머지 시간에 누적되는 변경 사항의 수를 적게 유지해줍니다.

코드형 파이프라인

일반적인 AWS 팀에서는 팀의 다양한 마이크로서비스와 소스 유형(애플리케이션 코드, 인프라 코드, OS 패치 등)을 배포하기 위해 여러 파이프라인을 사용합니다. 각 파이프라인은 갈수록 늘어나는 리전과 가용 영역에 맞추어 여러 배포 단계로 구성됩니다. 즉, 팀이 파이프라인 시스템, 배포 시스템 및 경보 시스템에서 관리해야 하는 구성이 많고, 최신 모범 사례와 새로운 리전 및 가용 영역을 지원하기 위해 많은 작업을 수행해야 합니다. 지난 몇 년간 저희는 안전한 최신 파이프라인을 일관되고 쉬운 방식으로 구성하기 위해 이 구성을 코드로 모델링하는 '코드형 파이프라인'이라는 방식을 도입했습니다. 내부 코드형 파이프라인 도구는 중앙에서 관리되는 리전 및 가용 영역 목록에서 새로운 리전 및 가용 영역을 가져와 AWS 전반의 파이프라인에 간단히 추가합니다. 또한 이 도구를 사용하면 팀이 상속을 통해 팀의 파이프라인 전반에 공통적으로 적용되는 구성(예: 각 웨이브에 포함되는 리전 및 각 웨이브의 베이크 시간)을 상위 클래스에 정의하고 모든 마이크로서비스 파이프라인 구성을 모든 공통 구성을 상속하는 하위 클래스로 정의하는 방식으로 파이프라인을 모델링할 수 있습니다.

결론

Amazon 은 배포의 안전성과 배포 속도 간의 적절한 균형을 맞추는 것을 목표로 장기간에 걸쳐 자동화된 배포 환경을 구축했습니다. 아울러 개발자가 배포를 관리하는 데 할애해야 하는 시간을 최소화하는 할 목적도 있었습니다. 광범위한 프로덕션 전 테스트, 자동 롤백, 단계별 프로덕션 배포 등의 방법을 활용하여 배포 안전을 보장할 자동화된 기능을 릴리스 프로세스에 구축함으로써 배포가 프로덕션 환경에 미칠 잠재적 영향을 최소화할 수 있습니다. 즉, 개발자가 프로덕션 환경에서 배포를 적극적으로 모니터링할 필요가 없습니다.

개발자는 완전 자동화된 파이프라인을 통해 코드 검토 기능을 사용하여 코드를 점검하고 프로덕션에 적용할 준비가 된 변경 사항을 승인합니다. 파이프라인이 변경 사항을 신중하고 안전하게 프로덕션 환경에 적용할 것임을 신뢰할 수 있기 때문에, 소스 코드 리포지토리에 변경 사항이 병합되고 나면 개발자는 다음 작업으로 넘어가고 배포에 대해서는 신경 쓸 필요가 없습니다. 자동화된 파이프라인은 안전과 속도의 균형을 유지하면서 하루에 여러 번 프로덕션 환경에 지속적으로 배포하는 작업을 처리합니다. 이 같은 지속적 전달 방식을 코드에 모델링함에 따라 AWS 서비스 팀은 코드 변경 사항을 자동으로 안전하게 배포하는 파이프라인을 손쉽게 설정할 수 있게 되었습니다.