지속적 전달을 통한 신속한 배포

Mark Mansour

지속적 개선과 소프트웨어 자동화

10 여 년 전에 Amazon 은 팀에서 아이디어를 얼마나 빠르게 고품질 프로덕션 시스템에 구현해내는지 알아보기 위한 프로젝트에 착수했습니다. 그 결과, 소프트웨어 처리량을 측정하여 작업 실행 속도를 높일 수 있었습니다. 코드를 프로덕션 환경에 체크인하는 데 평균적으로 16 일이 걸리는 것으로 나타났습니다. Amazon 의 팀에서는 새로운 아이디어가 나오면 보통 하루 반 만에 코드를 작성해 아이디어를 구현합니다. 새 코드를 빌드하고 배포하는 데는 한 시간도 채 걸리지 않습니다. 거의 14 일이나 되는 나머지 시간은 팀원이 빌드를 시작하고 배포를 수행하고 테스트를 실행하기를 기다리면서 보냅니다. 프로젝트 마지막에 사후 체크인 프로세스를 자동화하여 실행 속도를 높이자고 제안했습니다. 품질을 유지하거나 더 높이면서 지연을 없애는 것이 목표였습니다.

이 제안의 핵심은 실행 속도를 높이기 위한 지속적 개선 프로그램이었습니다. 실행 속도를 높이기로 한 결정은 "최고 수준의 표준에 대한 고집"이라는 경영 원칙에 따라 내려졌습니다. 이 원칙은 높은 수준의 표준을 엄격히 고수하면서 지속적으로 기준을 높이고 고품질의 제품, 서비스, 프로세스를 제공하는 것을 의미합니다. Amazon 의 경영 원칙은 Amazon 의 비즈니스 방식, 경영진의 리드 방식, 고객 중심의 결정 방식을 말해줍니다.

당시 Amazon 은 이미 소프트웨어 엔지니어들의 생산성을 높이는 소프트웨어 개발 도구를 만들어 놓았습니다. 중앙 집중식으로 호스팅되는 Brazil 이라는 빌드 시스템을 자체적으로 구축한 상태였습니다. 이 시스템은 서버에서 일련의 명령을 실행하여 배포 가능한 아티팩트를 생성합니다. 현재 Brazil 은 소스 코드 변경을 인식하지 못합니다. 사람이 빌드를 시작해야 합니다. 또한 Amazon 은 Apollo 라는 자체 배포 시스템을 사용하고 있었는데, 배포를 시작하는 프로세스의 일부로서 이 시스템에 빌드 아티팩트가 업로드되어야 했습니다. 업계에서 나타나고 있었던 지속적 전달 개념에 관심을 두면서 이에 착안해 Brazil 과 Apollo 간의 소프트웨어 제공 프로세스를 자동화하는 Pipelines 라는 자체 시스템을 구축했습니다.

Pipelines: Amazon 의 지속적 배포 도구

소수의 팀을 대상으로 소프트웨어 제공 프로세스를 자동화하는 파일럿 프로그램을 시작했습니다. 프로그램이 완료될 당시 헤드라인 파일럿 팀은 체크인 단계에서 프로덕션 단계까지 걸리는 전반적인 시간을 90% 단축한 상태였습니다.

이 프로젝트로 파이프라인 개념이 Amazon 팀에서 소프트웨어를 고객에게 릴리스하는 데 필요한 모든 단계를 정의하는 데 효과적이라는 것이 검증되었습니다. 파이프라인의 첫 단계는 아티팩트를 빌드하는 것입니다. 그런 다음 파이프라인에서는 아티팩트가 모든 고객에게 릴리스될 때까지 일련의 단계에 따라 해당 빌드 아티팩트를 실행합니다. Amazon 에서는 파이프라인을 사용하여

새로운 코드 변경 사항이 고객에게 부정적인 영향을 미칠 위험을 줄입니다. 파이프라인의 각 단계는 빌드 아티팩트에 결함이 없다는 확신을 높여주어야 합니다. 결함이 있는 상태로 프로덕션 환경에 배포될 경우에는 최대한 신속하게 정상 상태로 되돌릴 수 있어야 합니다.

Pipelines 을 사용하기 시작할 당시에는 애플리케이션당 하나의 릴리스 프로세스만 모델링할 수 있었습니다. 이 제한은 팀의 릴리스 프로세스에서 일관성, 표준화 및 간소화를 강화해주었습니다. 이를 통해 고객에게 노출되는 결함을 줄일 수 있었습니다. 파이프라인을 사용하기 전에는 팀마다 버그 픽스와 주요 기능 릴리스를 위해 여러 릴리스 프로세스를 실행해야 했습니다. 파일럿에 참여한 팀을 통해 자동화된 제공 방식의 성공을 지켜본 다른 팀들도 일관성을 높이기 위해 수작업으로 관리하던 릴리스 프로세스를 파이프라인으로 마이그레이션하기 시작했습니다. 다양한 릴리스 프로세스를 사용하던 팀들이 이제는 모두 표준화된 단일 프로세스를 이용합니다. 또한 릴리스 프로세스를 도구로 전환하는 과정에서 팀원들이 기존 방식을 다시 검토하고 프로세스를 간소화할 방법을 찾아내는 경우가 많았습니다.

Pipelines 을 사용하기 시작할 당시에는 애플리케이션당 하나의 릴리스 프로세스만 모델링할 수 있었습니다. 이 제한은 팀의 릴리스 프로세스에서 일관성, 표준화 및 간소화를 강화해주었습니다. 결함도 줄어들었습니다. 그러자 여러 팀들이 소프트웨어를 릴리스하고 완전 자동화된 릴리스 방식으로 전환하기 위해 파이프라인을 사용한다는 목표에 동참하게 되었습니다. 하지만 일부 조직에서는 품질을 측정하는 우리의 방식이 아무런 테스트도 제공하지 않은 채 팀들이 무작정 릴리스 프로세스를 자동화하게 만들고 있음을 알게 되었습니다.

“테스트를 얼마나 해야 충분한가?”라는 질문의 답은 순전히 주관적인 판단입니다. 이 질문에 답하려면 팀 운영의 제반 상황을 이해해야 합니다. 이 상황을 해결하기 위해 책임 의식이라는 또 다른 경영 원칙을 차용했습니다. 이 원칙은 장기적으로 생각하고 단기적인 성과를 위해 장기적인 가치를 저해하지 않는 것을 말합니다. Amazon 의 소프트웨어 팀은 높은 수준의 테스트 기준을 적용하며 테스트에 공을 많이 들입니다. 제품을 담당한다는 것은 해당 제품의 결함에 따른 결과까지 책임을 진다는 의미이기 때문입니다. 고객에게 영향을 미치는 문제가 발생할 경우 소규모의 단일 스레드 소프트웨어 팀의 팀원이 실시간으로 사안을 처리하고 문제를 해결해야 합니다. 높아지는 실행 속도와 프로덕션 환경에서 문제에 대응하는 것 사이에서 형성되는 긴장 상태는 팀이 충분한 테스트를 거치도록 하는 동기 부여가 됩니다. 하지만 테스트에 너무 과하게 투자할 경우 다른 경쟁업체가 시장을 선점하여 성과를 거두지 못할 수 있습니다. 이에 Amazon 에서는 늘 비즈니스를 저해하는 요인이 되지 않으면서 소프트웨어 릴리스 프로세스를 개선할 방법을 찾고 있습니다.

우리가 직면했던 다른 문제로는 팀원들 간에 소프트웨어 릴리스 모범 사례를 서로 배우지 않고 있다는 점이었습니다. 단일 스레드 팀들에게는 업무의 자율성이 주어졌기 때문에 엔지니어들이 배포 문제를 독립적으로 해결하고 있었습니다. 소프트웨어 릴리스 요구 사항을 해결할 솔루션을 찾아내면 메일링 리스트, 운영 회의, 기타 커뮤니케이션 채널을 통해 그 기법을 다른 엔지니어들에게 알리게 됩니다. 그런데 이 같은 커뮤니케이션 스타일에는 두 가지 문제가 있었습니다. 첫째, 이러한 채널은 가능한 한 노력하는 방식의 커뮤니케이션 채널입니다. 즉, 모든 사람이 새 기법을 배우게 되는 것은 아닙니다. 둘째, 리더들이 팀원들에게 새 모범 사례를 도입하도록 독려하는 방식은 팀원들이 실제로 모범 사례를 도입하는 데 필요한 작업을 했는지 파악할 길이 없습니다. 우리가 경험을 통해 배운 모범 사례를 모든 엔지니어가 접할 수 있도록 하고 리더들에게는 주의를 요하는 파이프라인을 식별하는 능력을 주어야 한다는 사실을 알게 되었습니다.

이를 위한 솔루션은 소프트웨어를 빌드하고 릴리스하는 데 사용하는 도구에 모범 사례를 점검하는 기능을 추가하여 학습을 기계화하는 것이었습니다. 특정 조직의 모범 사례가 다른 조직에는 안 맞을 수도 있다는 점을 감안해 조직별로 이 점검 항목을 구성할 수 있도록 했습니다. 조직 차원의 모범 사례 점검 기능은 리더에게 비즈니스 요구 사항에 맞추어 릴리스 프로세스를 조정할 수 있는 능력을 부여했습니다. 모범 사례를 장려하거나 적용하기를 원하던 리더들은 엔지니어들이 일상적으로 사용하는 도구 내에서 경고를 제공하는 것부터 시작해 그러한 목표를 실현할 수 있었습니다. 도구에 메시지를 표시함으로써 팀원이 모범 사례를 익히고 모범 사례가 적용되도록 거의 100% 보장할 수 있었습니다. 팀원들이 새로운 모범 사례를 익히고 논의할 시간을 줌으로써 조직에서는 모범 사례 점검을 반복 적용하고 개선할 기회를 얻을 수 있다는 사실을 알게 되었습니다. 궁극적으로 이는 모범 사례의 질이 높아지고 엔지니어들 사이에서 더 효과적으로 수용되는 결과로 이어졌습니다.

적용할 모범 사례는 체계적으로 선별했습니다. 최선임 엔지니어들이 릴리스가 제대로 작동하지 않는 일반적인 이유를 카탈로그로 만들었습니다. 이들은 릴리스를 정상 작동하도록 만들기 위한 단계를 찾아냈습니다. 그런 다음 그 목록을 바탕으로 모범 사례 점검 항목을 만들었습니다. 이 프로세스를 통해 새로운 소프트웨어 개정 버전이 추가적인 작업 없이 가용성을 저하시키지 않으면서 고객에게 즉각적으로 제공되기를 원하지만, 속도보다 가용성이 우선이고 엔지니어가 작업하기 쉽게 만드는 것이 우선이라는 사실을 알게 되었습니다.

결함이 고객에게 노출될 위험 줄이기

엔지니어라면 누구나 어느 시점에는 시스템 중 하나에 결함을 만들어내기 마련입니다. 파이프라인과 배포 시스템을 비롯한 Amazon 의 릴리스 프로세스는 그러한 결함을 최대한 빨리

찾아내 고객에게 영향을 미치지 않도록 방지할 수 있게 설계되어야 합니다. 릴리스 프로세스가 올바르게 구성되고 빌드 아티팩트가 의도대로 작동하도록 보장해야 합니다.

배포 방역: 배포 테스트의 기본 형태는 대부분 새로 배포된 아티팩트가 시작할 수 있고 작업에 응답할 수 있는 상태임을 보장합니다. 배포 후 워크플로의 일환으로서 새로 배포된 아티팩가 시작되어 트래픽을 처리하는지 확인하는 간단한 점검을 실행합니다. 예를 들어 AWS CodeDeploy AppSpec 파일에 수명 주기 이벤트 후크를 사용하여 배포를 중지, 시작 및 검증하는 간단한 스크립트를 트리거합니다. 또한 고객 트래픽을 지원하기에 용량이 충분한지도 점검합니다. 항상 고객에게 서비스를 제공하는 데 충분한 용량이 확보되도록 보장하기 위해 CodeDeploy 의 최소 상태 호스트와 같은 기법을 개발했습니다. 마지막으로, 배포 엔진이 장애를 감지할 수 있다면, 변경 사항을 롤백하여 고객에게 결함이 나타나는 시간을 최소화해야 합니다.

프로덕션 단계 전에 테스트: Amazon 의 모범 사례 중 하나는 유닛, 통합 및 프로덕션 전 테스트를 자동화하고 이들 테스트를 파이프라인에 추가하는 것입니다. 로드 테스트와 보안 테스트는 반드시 수행하는 것을 원칙으로 하고 있어서 이들 테스트는 꼭 파이프라인에 추가하는 편입니다. 유닛 테스트란 빌드 시스템에 대해서 수행하는 테스트로, 스타일 점검, 코드 적용 범위, 코드 복잡성 등을 테스트하는 것입니다. 통합 테스트는 장애 주입, 자동화된 브라우저 테스트 등의 모든 오프박스 테스트를 포함하는 통합 테스트라고 할 수 있습니다. 유닛 테스트와 통합 테스트에 대해 설명하는 훌륭한 문서가 많으므로 여기서는 자세히 다루지 않겠습니다.

유닛 테스트와 통합 테스트는 빌드 아티팩트가 올바르게 동작하는지 확인하는 것을 목표로 합니다. 검증을 많이 수행할수록 결함이 고객에게 노출될 위험이 적어집니다. 제품을 고객에게 제공하기까지 걸리는 시간을 단축하기 위해 최대한 릴리스 프로세스 초기에 결함을 찾아내려고 합니다. 일반적으로 이는 테스트 규모가 작고 빠를수록 변경 사항에 따른 문제에 대한 피드백을 더 빨리 받을 수 있음을 의미합니다.

Amazon 에서는 *프로덕션 전 테스트*라는 기법도 사용합니다. 프로덕션 전 환경은 변경 사항을 프로덕션 환경에 배포하기 전에 마지막 테스트가 이루어지는 곳입니다. 프로덕션 전 환경 테스트에서는 시스템의 프로덕션 구성을 사용하므로 프로덕션 시스템과 똑같이 작동합니다. 이 방식은 두 가지 이점이 있습니다. 첫째, 프로덕션 전 환경 테스트는 프로덕션 구성을 테스트하여 서비스가 프로덕션 데이터 스토어를 비롯한 모든 프로덕션 리소스에 정상적으로 연결할 수 있는지 확인합니다. 둘째, 시스템이 필요한 프로덕션 서비스의 API 와 올바르게 상호 작용하는지 확인합니다. 프로덕션 전 환경은 서비스를 소유한 팀에서만 사용하며 고객에게 트래픽을 전송하지 않습니다. 프로덕션 전 테스트를 실행하면 동일한 코드와 구성이 프로덕션 환경에서도 정상적으로 작동할 것이라는 데 대한 신뢰도가 높아집니다.

프로덕션 환경에서 검증: 저희는 코드를 모든 고객에게 동시에 릴리스하지 않습니다. 모든 고객에게 동시에 결함을 릴리스하면 결함이 미치는 영향의 범위가 너무 커집니다. 대신 서비스의 완전히 독립된 인스턴스인 셀에 배포합니다. 변경 사항을 첫 번째 셀의 첫 고객 집단에 배포할 때는 각별한 주의를 기울입니다. 소수의 고객에게만 새로운 변경 사항이 노출되도록 하고 새 코드가 제대로 작동하는지에 대한 피드백을 받습니다. 카나리아 배포 후 서비스에서 발생하는 오류의 양을 모니터링합니다. 그리고 오류 발생률이 높아지면 변경 사항을 자동으로 롤백합니다. 예를 들어 부정적인 데이터 포인트 없이 3,000 개의 긍정적인 데이터 포인트가 수집되면 그때 비로소 배포를 계속 진행합니다.

자동화된 테스트에서 놓치는 사용 사례가 있으면 문제가 나타날 수 있습니다. 그래서 저희는 자동 테스트와 수동 테스트를 막론하고 체계적이고 반복적인 테스트를 통해 모든 오류를 잡아내기 위해 최선을 다합니다. 하지만 아무리 최선을 다하더라도 잡아내지 못하는 결함이 있을 수 있습니다. 테스트 자체를 테스트하기 위해 새로운 변경 사항을 프로덕션 환경에 정해진 기간 동안 남겨 두고 팀원 이외의 사용자가 문제를 찾아내는지 봅니다. 저희는 변경 사항을 프로덕션에 그대로 둘지 여부 또는 카나리아 배포 후 나머지 배포 그룹에 배포할 때까지 얼마나 기다려야 할지 논의하는 데 많은 시간을 투자했습니다. 많은 팀이 일정 시간 동안 기다리면서 긍정적인 데이터 포인트를 수집한 후 배포 루틴을 통해 처리하기로 결정했습니다. 파이프라인 대기 시간은 팀에 의해 크게 좌우됩니다. 몇 시간을 기다리는 팀도 있고 몇 분만 기다리는 팀도 있습니다. 문제의 영향이 크고 해결하는 데 시간이 많이 걸릴수록 릴리스 프로세스는 느려집니다.

첫 번째 셀에서 확신을 얻고 나면 완전히 릴리스될 때까지 새 코드 변경 사항을 점진적으로 더 많은 고객에게 노출합니다. 카나리아 배포와 마찬가지로 첫 번째 새로운 셀에 배포한 결과에 대한 확신을 얻을 때까지 기다린 후 다음 셀로 진행합니다. 빌드 아티팩트의 신뢰도가 높아짐에 따라 코드 변경 사항을 검증하는 데 들이는 시간을 줄여나갑니다. 이 같은 과정을 통해 체크인 단계부터 첫 번째 프로덕션 고객에게 배포될 때까지 최대한 시간을 단축하는 것을 목표로 한 패턴이 만들어집니다. 하지만 일단 프로덕션 환경에서 배포를 시작한 후에는 서서히 고객에게 새 코드를 릴리스하면서 신뢰도를 더 높이는 작업을 진행하고 나머지 배포 작업의 속도를 점진적으로 높여갑니다.

프로덕션 시스템이 고객의 요청을 지속적으로 서비스하는지 확인하기 위해 시스템에서 가상트래픽을 생성합니다. 서비스가 제대로 작동하지 않을 경우 신속한 피드백을 받을 목적으로, 적어도 1 분마다 한 번씩 가상 테스트를 실행합니다. 가상 테스트는 실행 중인 프로세스가 정상 상태인지 확인할 수 있고 모든 종속 구성 요소가 테스트되도록 가상 테스트를 설계합니다. 이를 위해 모든 공용 API 를 테스트해야 하는 경우가 많습니다.

소프트웨어 릴리스 시점 제어: 소프트웨어 릴리스의 안전성을 관리하기 위해 변경 사항이 파이프라인을 거치는 속도를 제어할 수 있는 메커니즘을 구축했습니다. 즉, 지표, 시간대, 안전 점검을 활용해 소프트웨어 릴리스 시점을 제어합니다.

지표의 변화를 기준으로 경보가 발생할 경우 배포를 방지하도록 파이프라인을 구성할 수 있습니다. 지표를 광범위하게 사용하고 시스템 상태, 셀 상태, 가용 영역 및 리전 등 떠올릴 수 있는 대부분의 항목에 대한 경보를 적용합니다. 중요 지표에서 경보가 트리거되면 코드 배포를 중단하도록 파이프라인을 구성합니다. 하지만 시스템의 경보를 해결하기 위해 팀원이 픽스를 배포해야 하는 경우도 있습니다. 이 시나리오에서는 변경 사항이 파이프라인을 통해 진행되지 못하도록 하는 팀이 경보를 무시할 수 있게 합니다.

파이프라인은 변경 사항이 파이프라인을 통해 진행되도록 허용되는 시간대를 지정할 수 있습니다. 팀에서는 변경 사항이 고객에게 노출되는 것을 제한할 수 있는 시간대를 자체적으로 찾을 수 있습니다. AWS 팀들은 배포로 인해 발생하는 문제에 신속하게 대응하여 문제를 해결할 수 있는 사람들이 많이 있는 시간대에 소프트웨어를 릴리스하는 것을 선호합니다. 이를 실현하기 위해 일반적으로 팀에서는 업무 시간 중에만 배포하도록 시간대를 정합니다. 고객 트래픽이 적은 시간에 소프트웨어를 릴리스하려고 하는 Amazon 팀들도 있습니다. 이 같은 시간대는 필요한 경우 재정의할 수 있습니다.

또한 빌드 아티팩트의 콘텐츠에 따라 파이프라인을 중지하는 기능도 있습니다. 예를 들어 알려진 불량 패키지 또는 특정 Git 참조가 포함된 빌드 아티팩트를 차단할 수 있습니다. 이 기능은 패키지의 변경 사항에 성능 저하 문제가 있음을 발견했을 때 사용하고 있습니다. 단순히 패키지 리포지토리에서 패키지를 제거한다면, 결함 있는 패키지가 이미 포함된 파이프라인이 여전히 잘못된 변경 사항을 고객에게 배포할 것입니다.

Amazon 이 지금의 실행 속도를 실현한 비법

저희는 팀들의 자동화 도입 욕구가 크다는 것을 알게 되었습니다. 고객의 편의성을 높이고 지속적 전달을 안정적으로 만드는 기능을 빌드하여 고객에게 릴리스하는 데 모두가 매우 적극적입니다. 자동화가 번거롭고 오류가 발생하기 쉬우며 노동 집약적인 수작업을 없앰으로써 엔지니어에게 여유 시간을 확보해준다는 것을 확인했습니다. 지속적 배포가 품질에 긍정적 영향을 미친다는 사실도 확인했습니다. 자동화가 역효과를 쉽게 파악할 수 있게 하여 팀들이 한 번에 하나씩 변경 사항을 자주 릴리스할 수 있도록 한다는 것을 확인했습니다.

새로운 시스템의 경우 일반적으로 테스트할 노출 영역을 대부분의 팀원이 잘 알기 때문에 일부 수동 테스트는 수월하게 진행할 수 있습니다. 하지만 시스템이 갈수록 복잡해지고 팀원이 바뀌면서

자동화가 점점 중요해집니다. 우리는 그러한 변경 사항을 고객에게 제공하는 프로세스를 수동으로 관리하는 데 주력하는 것이 아니라 고객에게 부가가치를 제공하는 데 집중할 수 있다는 점에서 자동화를 좋아합니다.

수년간 Amazon 은 고객에게 소프트웨어를 릴리스하는 속도와 그러한 릴리스의 안전성에 초점을 맞추면서 지속적 개선 프로그램을 진행해왔습니다. 이 문서에서 다룬 모든 위험 점검과 테스트가 처음부터 갖추어졌던 것은 아닙니다. 시간이 지나면서 위험을 포착하고 해결할 방법을 하나씩 찾아낸 것입니다.

지속적 개선 프로그램은 조직의 다양한 직급에 있는 비즈니스 리더들이 운영합니다. 따라서 각 비즈니스 리더가 비즈니스에 미치는 위험과 영향에 따라 소프트웨어 릴리스 프로세스를 조정할 수 있습니다. 지속적 개선 프로그램 중 일부는 Amazon 에서 많은 부분을 차지하는 여러 분야에 걸쳐 진행되고, 소규모 조직의 리더가 자체적인 프로그램을 진행하는 경우도 있습니다. 어떤 규칙이든 예외가 있기 마련입니다. Amazon 의 시스템에는 영구적이거나 일시적인 예외가 필요한 팀을 지연시키지 않기 위해 테스트 메커니즘을 적용하지 않도록 선택할 수 있는 옵션도 있습니다. 궁극적으로, Amazon 의 팀들은 소프트웨어의 동작에 대한 책임 의식을 갖고, 소프트웨어 릴리스 프로세스에 적절한 투자를 할 책임을 집니다.

먼저 문제를 진단한 후 해결하고 그 과정을 반복합니다. 이 작업을 지속적으로 수행하기 위해서는 점진적으로 수행하고 장기간에 걸친 개선을 평가해야 합니다. Amazon 에서 파이프라인을 처음 사용하기 시작할 당시 많은 팀이 지속적 배포가 과연 효과적일지 확신하지 못했습니다. 팀들이 도입에 나서도록 하기 위해 현재 릴리스 프로세스와 수작업 단계 등을 모두 파이프라인에 인코딩하도록 독려했습니다. 많은 팀에게 있어 파이프라인은 릴리스 프로세스를 통해 빌드 아티팩트를 자동으로 승격시키지 않는 시각적 인터페이스 역할을 했습니다. 확신이 커지면서 팀들은 파이프라인의 여러 단계에 서서히 자동화를 도입해 결국 파이프라인의 어떤 단계에서도 수동으로 작업을 트리거할 필요가 없게 되었습니다.

이제 현재로 뛰어넘어와서, Amazon 은 지금, 팀들이 새 코드를 작성하여 적용하는 프로세스를 완전 자동화하는 것을 목표로 하기에 이르렀습니다. Amazon 에 있어 자동화는 비즈니스의 성장을 지속적으로 도모할 수 있는 유일한 방법입니다.