제26장

건축적 교차점

이 책에서 지금까지 우리는 아키텍처가 지원해야 하는 핵심적인 특성을 식별하는 방법, 이러한 특성과 비즈니스 문제에 가장 적합한 아키텍처 스타일을 선택하는 방법, 효과적인 아키텍처 결정을 내리는 방법, 그리고 개발 팀이 아키텍처를 구현하도록 이끌고 안내하는 방법을 보여주었습니다. 그러나 아키텍처가 제대로 작동하려면 기술 및 비즈니스 환경의 다른 측면들과도 조화를 이루어야 합니다. 우리는 이러한 조화를 아키텍처의 교차점이라고 부릅니다.

이 장에서는 소프트웨어 아키텍처를 생성하거나 검증할 때 발생하는 몇 가지 중요한 교차점에 대해 논의합니다.

구현은 운영 특성, 아키
: 텍처 제약 조건 및 아키텍처의 내부 구조와 관련된 아키텍처적 고려 사항과 일치하는가?

인프라 측면에
: 서, 아키텍처의 구축 방식과 인프라가 확장성, 응답성, 내결함성, 가용성 등 운영상의 고려 사항과 부합하는지 확인해야 합니다.

데이터 토폴로지.
: 흔히 간과되는 부분 중 하나는 아키텍처와 데이터 토폴로지 및 데이터 유형 간의 상호 작용입니다. 시스템이 제대로 작동하려면 데이터 토폴로지(모놀리식, 도메인 데이터베이스, 서비스별 데이터베이스)가 아키텍처 스타일과 적절히 일치해야 합니다.

엔지니어링 관행은 개발팀
: 이 소프트웨어를 생성, 유지 관리 및 테스트하는 방식이 해당 아키텍처와 일치하는지 여부를 판단하는 데 중요합니다. 배포 파이프라인은 아키텍처 스타일과 일치하는지 확인해야 합니다.

팀 구성 방식은 아
: 키텍처에 상당한 영향을 미칠 수 있으며, 그 반대의 경우도 마찬가지입니다. 팀 구조가 아키텍처와 제대로 일치하지 않으면 개발 팀은 일반적으로 어려움을 겪게 되며, 가장 간단한 변경조차도 어려워합니다.

시스템 통합은 아키텍
: 처가 통신해야 하는 다른 시스템이나 서비스는 무엇인지 파악하는 데 중요한 요소입니다. 이러한 상호 작용에 주의를 기울이지 않으면 유지 관리, 신뢰성, 확장성, 응답성, 가용성 등의 운영 특성 측면에서 심각한 결과를 초래할 수 있습니다.

e-엔터프라이
: 즈 아키텍처가 조직 및 기업 전반의 프레임워크, 관행, 기본 원칙 및 표준과 일치합니까?

비즈니스 환경과 문제 영역에 아키
: 텍처가 제대로 부합하는지 확인해야 합니다. 아키텍트들은 이 중요한 접점을 간과하는 경우가 너무 많으며, 그 결과 아키텍처가 비즈니스의 목표나 요구 사항을 충족하지 못하게 됩니다.

생성형 인공지능
: 대규모 언어 모델(LLM)의 사용 증가가 아키텍처에 어떤 영향을 미칠까요? 더 많은 기업들이 시스템에 생성형 AI를 활용함에 따라 이 두 분야의 접점은 빠르게 중요해지고 있습니다.

다음 섹션에서는 이러한 교차점에 대해 더 자세히 설명합니다.

아키텍처 및 구현

소프트웨어 아키텍처의 첫 번째 법칙은 소프트웨어 아키텍트가 어떤 질문에든 가장 흔히 하는 대답이기도 합니다. 바로 "상황에 따라 다릅니다." 두 번째로 흔한 대답은 "그건 구현 세부 사항입니다."입니다. 소프트웨어 아키텍처가 목표를 달성하지 못할 때, 종종 이 두 번째 대답이 원인으로 지목되곤 합니다.

아키텍처가 제대로 작동하려면 구현, 즉 소스 코드가 설계와 일치해야 하며, 다음 세 가지 사항을 고려해야 합니다. 첫째, 아키텍처의 운영 관련 고려 사항(예: 내결함성, 응답성, 확장성 등), 둘째, 내부 구조, 셋째, 제약 조건입니다. 이 섹션에서는 이 세 가지 사항을 차례로 다룹니다.

운영적 고려 사항은 이 책 1부 에서 중

점적으로 다룬 아키텍처적 특성으로 , 모든 소프트웨어 아키텍처의 기반을 형성하며, 당면한 비즈니스 문제를 해결하기 위해 아키텍처가 반드시 지원해야 하는 부분입니다.

건축적 특징은 건축적 결정의 주요 동인입니다(21 장 에서 논의됨).

그렇다면 아키텍처와 구현 방식이 시스템 운영상의 문제점을 해결하는 방식에서 불일치를 보인다는 것은 무엇을 의미할까요? 예를 들어, 수천 명에서 최대 50만 명의 동시 접속 고객을 지원해야 하는 새로운 주문 입력 시스템을 개발하는 아키텍트라고 가정해 보겠습니다. 18장 354페이지 의 "스타일 특징" 에 나와 있는 별점 평가를 바탕으로 , 이 시스템의 높은 확장성과 탄력성 요구 사항을 고려하여 마이크로서비스 아키텍처 스타일을 선택했습니다. 그러나 구현 과정에서 개발팀은 서비스의 경계 컨텍스트가 너무 엄격하게 설정되어 있어(18장 331페이지의 "경계 컨텍스트" 참조), 주문 접수 서비스가 재고 데이터베이스에 직접 접근할 수 없다는 것을 발견했습니다 . 대신 , 고객 이 구매하려는 품목의 현재 재고를 가져오기 위해 재고 서비스 를 동기적으로 호출해야 합니다 . 이러한 동기 호출은 두 서비스를 강하게 결합시킬 뿐만 아니라 시스템 응답 속도를 크게 저하시킵니다.

개발팀은 그림 26-1 에서 보는 것처럼 이러한 서비스 간에 메모리 내 복제 캐시를 사용하기로 결정했습니다. 데이터는 각 서비스 인스턴스의 내부 메모리에 저장되며, 캐싱을 통해 백그라운드에서 항상 동기화됩니다. 이러한 목적에 적합한 캐싱 제품으로는 Apache Ignite 와 Hazelcast가 있습니다. 예를 들어, 재고 관리 서비스는 품목 ID와 현재 재고 수량을 저장하는 쓰기 가능한 메모리 내 캐시를 가지며, 주문 처리 서비스의 각 인스턴스는 내부 메모리에 해당 캐시의 복제된 읽기 전용 버전을 갖게 됩니다. 메모리 내 복제 캐시를 사용하면 서비스 간의 결합도가 낮아지고 응답성이 크게 향상됩니다.

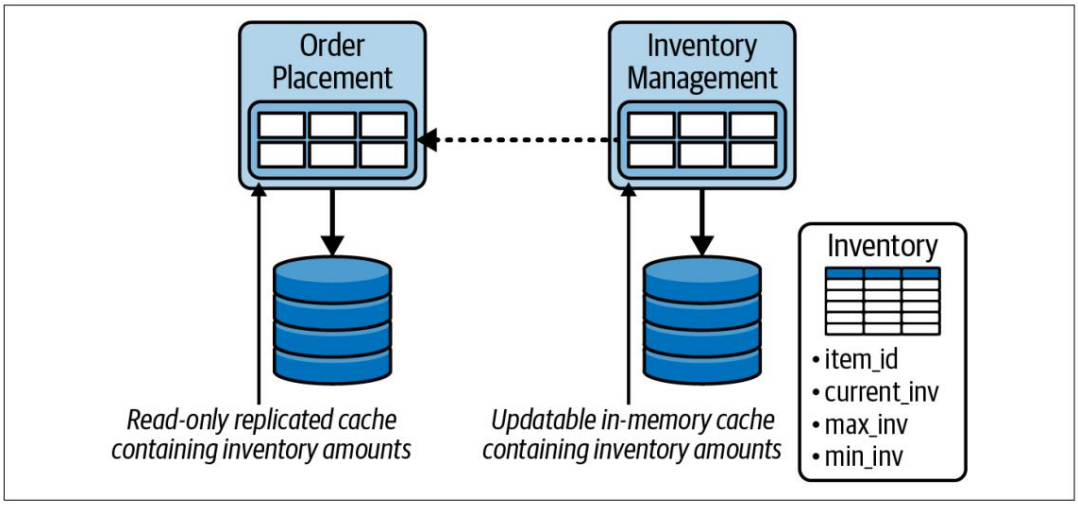

그림 26-1. 개발팀은 서비스 간 인메모리 복제 캐싱을 사용하기로 결정했는데, 이로 인해 서비스 확장 시 메모리 부족 오류가 발생했습니다.

정식 출시 후 동시 접속 사용자 수가 증가함에 따라 부하를 처리하기 위해 각 서비스의 인스턴스 수가 늘어납니다. 하지만 동시 접속 사용자 수가 약 8만 명에 도달하면 내부 캐시의 메모리 요구량이 너무 높아져 모든 가상 머신에서 메모리 부족 오류가 발생하고 시스템이 다운됩니다.

이 시나리오에서는 아키텍처와 구현이 서로 어긋나 있습니다. 아키텍처는 높은 수준의 확장성과 탄력성을 지원하는 데 중점을 두는 반면, 구현은 응답성과 서비스 분리에 중점을 둡니다. 두 팀 모두 좋은 결정을 내렸지만, 그 목표는 서로 달랐습니다.

구조적 무결성 8장 에서 배웠 듯

이 논리적 구성 요소는 모든 시스템의 기본 구성 요소이며 논리적 아키텍처를 형성합니다. 이러한 논리적 구성 요소는 일반적으로 소스 코드 저장소의 디렉터리 구조(또는 프로그래밍 언어에 따라 네임스페이스)를 통해 표현됩니다. 논리적 아키텍처는 시스템의 작동 방식과 시스템 구성 요소 간의 상호 작용 방식을 설명하므로 소스 코드 구조가 논리적 아키텍처와 일치하는 것이 매우 중요합니다.

적절한 지침, 지식 및 관리 체계가 없다면 개발자는 시스템의 논리적 아키텍처를 무시하고 시스템 무결성에 미치는 영향을 고려하지 않고 마음대로 디렉터리 구조와 네임스페이스를 생성하기 쉽습니다.

이러한 불일치로 인해 유지 관리, 테스트 및 배포가 어려운 아키텍처가 생성되고 결과적으로 신뢰성이 떨어지고 그림 26-2에 나타난 논리적 아키텍처와 같이 새로운 기능에 맞게 발전시키거나 적응하기가 더 어려워집니다.

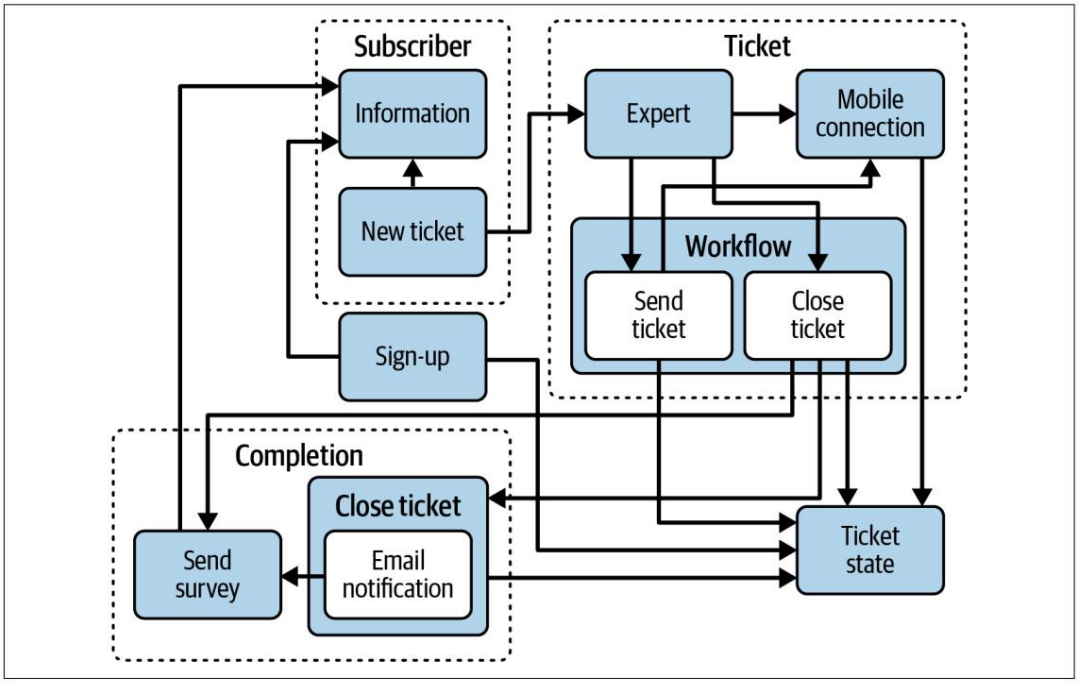

그림 26-2. 거버넌스와 정렬이 부족한 내부 논리 아키텍처의 예

소스 코드 구조가 논리적 아키텍처와 일치하도록 하려면 Java 플랫폼용 ArchUnit , .NET 플랫폼용 ArchUnitNet 및 NetArchTest , Python용 PyTestArch , TypeScript 및 JavaScript용 TSArch 와 같은 자동화된 거버넌스 도구를 사용하는 것이 좋습니다 . 이러한 자동화 도구는 아키텍트와 개발팀 간의 원활한 소통 및 협업과 함께 그림 26-3에서처럼 아키텍처에 적절히 부합하는 구현을 만들어냅니다 .

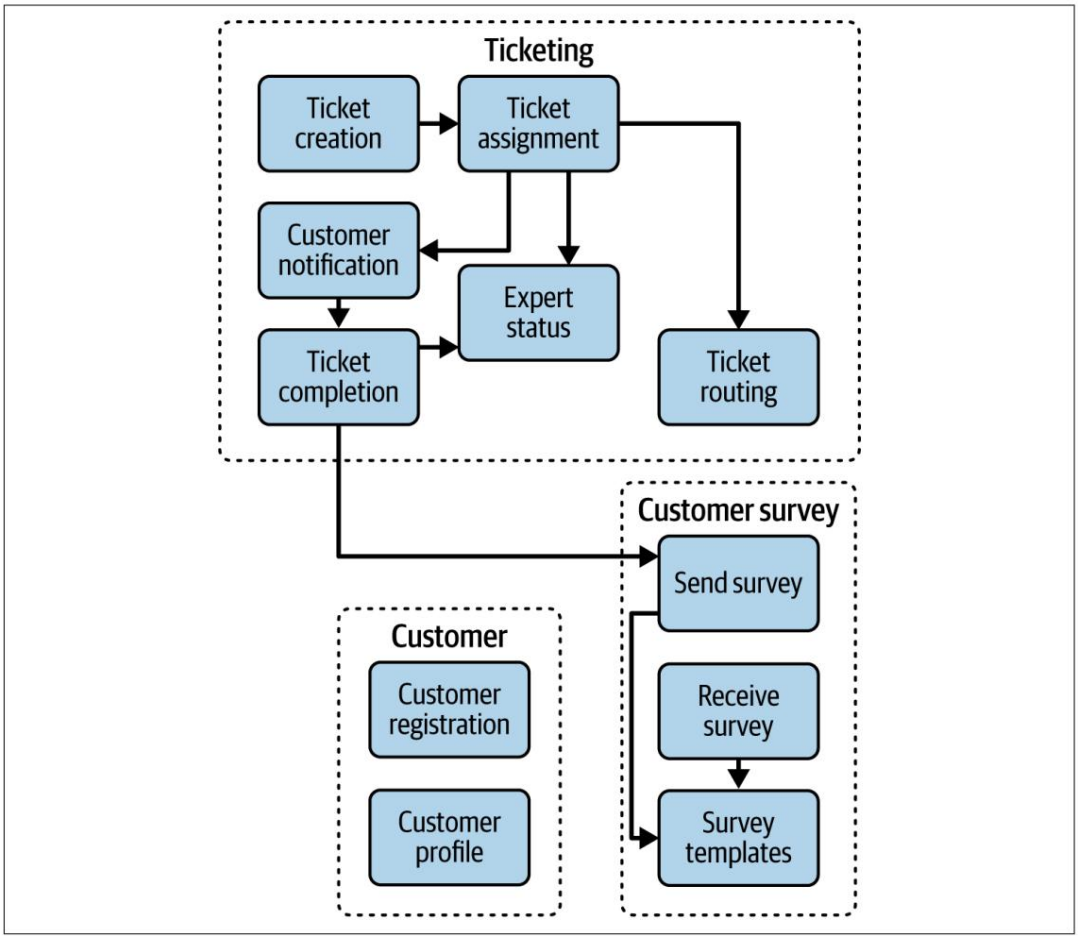

그림 26-3. 적절한 거버넌스와 정렬에 기반한 내부 논리 아키텍처의 예

그림 26-2 와 그림 26-3을 비교해 보십시오 . 그림 26-3 의 아키텍처가 그림 26-2 에 나타난 정렬되지 않은 아키텍처보다 유지보수성, 테스트 용이성, 배포 용이성, 신뢰성, 적응성 및 확장성이 훨씬 뛰어나다는 것을 알 수 있습니다. 이 비교는 이러한 구현 정렬의 중요성을 보여줍니다.

건축적 제약

제약 조건이란 아키텍처의 목표를 달성하기 위해 필요한 특정 제한 사항(예: 통신을 REST 방식으로만 제한하거나 특정 유형의 데이터베이스만 사용)을 설명하는 지배적인 규칙 또는 원칙입니다. 시스템 구현이 이러한 제약 조건을 준수하지 않으면 아키텍처는 제대로 작동하지 않습니다. 따라서 소프트웨어 아키텍트의 역할 중 하나는 아키텍처의 제약 조건을 파악하고 전달하는 것입니다.

이러한 특정 교차점이 무엇을 의미하는지 설명하기 위해, 매우 제한된 예산과 촉박한 기한 내에 새로운 시스템을 출시하려는 기업을 생각해 보겠습니다. 이 기업은

데이터베이스 구조에 많은 변경 사항이 예상되며, 이러한 변경 사항을 최대한 신속하게 처리해야 합니다. 이 경우, 단순성, 비용 효율성 및 기술적 파티셔닝 덕분에 전통적인 계층형 아키텍처(10장 참조) 가 매우 적합합니다. 이 방식은 계층을 분리하기 때문에 데이터베이스 변경 사항을 특정 계층에만 국한하여 처리할 수 있으므로 변경 작업이 더 쉽고 빨라집니다.

계층형 아키텍처가 이 특정 비즈니스 문제에서 제대로 작동하려면 아키텍트는 다음과 같은 제약 조건을 정의해야 합니다.

- 모든 데이터베이스 로직은 영속성 계층에 있어야 합니다. • 프레젠
  테이션 계층은 영속성 계층에 직접 접근할 수 없으며, 영속성 계층을 거쳐야 합니다.
  간단한 쿼리조차도 모든 계층을 거쳐 처리됩니다.

이러한 제약 조건은 데이터베이스 로직이 아키텍처 전체에 퍼지는 것을 방지하고, 물리적 데이터베이스 구조 변경(예: 테이블 삭제 또는 열 이름 변경)이 영속성 계층 외부의 코드에 영향을 미치지 않도록 하기 위해 필요합니다.

이제 UI 개발자들이 데이터베이스에 직접 접근하는 것이 더 빠르다고 판단하여 아키텍처를 그렇게 구현했다고 가정해 봅시다. 게다가 백엔드 개발자들은 비즈니스 로직과 데이터베이스 로직을 함께 두는 것이 유지보수와 테스트가 훨씬 쉽다는 것을 깨닫고, 기존의 제약 조건을 무시하고 이러한 로직들을 아키텍처의 비즈니스 계층에 결합시킵니다. 이렇게 구현된 방식은 아키텍처의 제약 조건을 준수하지 않기 때문에 데이터베이스 변경 시 모든 계층의 코드에 영향을 미치고, 수정 시간이 지나치게 오래 걸리며, 결국 시스템이 비즈니스 목표를 달성하지 못하게 됩니다.

건축 설계 도구는 건축 설계 제약 조건을 관리하는 데에도 유용합니다.

건축 및 인프라

지난 수십 년 동안 소프트웨어 아키텍처의 범위는 점점 더 많은 책임과 관점을 포괄하도록 확장되었습니다. 2000년대 중반까지만 해도 아키텍처와 운영 간의 관계는 계약적이고 형식적이었으며, 관료주의적인 측면이 강했습니다. 대부분의 기업은 자체 운영을 호스팅하는 복잡성을 피하기 위해 운영을 제3자에게 아웃소싱했으며, 가동 시간, 확장성, 응답성 및 기타 중요한 특성에 대한 SLA를 요구했습니다. 그러나 오늘날 마이크로서비스와 같은 아키텍처 스타일은 과거에는 오로지 운영적인 측면에만 국한되었던 특성들을 자유롭게 활용합니다. 예를 들어, 탄력적인 확장성은 과거에는 공간 기반 아키텍처(16장 참조)에 내장되어 있었지만, 오늘날 마이크로서비스는 아키텍트와 DevOps 팀 간의 긴밀한 협업을 통해 이를 훨씬 수월하게 처리합니다.

역사: 펫츠닷컴이 어떻게 탄력적 확장의 개념을 탄생시켰을

까? 사람들은 흔히 현재의 기술적 능력(예: 탄력적 확장)이 어느 날 갑자기 어떤 똑똑한 개발자에 의해 발명된 것이라고 생각합니다. 하지만 실제로는 최고의 아이디어는 종종 시행착오를 통해 탄생합니다. 펫츠닷컴이 바로 그 초기 사례입니다. 이 전자상거래 사이트는 1998년경에 등장하여 반려동물 용품 업계의 아마존닷컴이 되기를 꿈꿨습니다. 뛰어난 마케팅 부서는 매력적인 마스코트를 만들어냈는데, 바로 마이크를 들고 재치 넘치는 말을 하는 양말 인형이었습니다. 이 마스코트는 슈퍼스타가 되어 퍼레이드와 전국 스포츠 행사에도 등장했습니다.

안타깝게도 펫츠닷컴 경영진은 마스코트에 모든 돈을 쏟아붓고 인프라 구축에는 투자하지 않은 것으로 보입니다. 주문이 폭주하기 시작하자 그들은 제대로 대비하지 못했습니다. 웹사이트는 느려지고, 거래는 누락되고, 배송은 지연되는 등 최악의 상황이 벌어졌습니다. 재앙에 가까운 크리스마스 시즌을 보낸 직후, 펫츠닷컴은 문을 닫고 유일하게 남은 가치 있는 자산인 마스코트를 팔아넘겼습니다.

펫츠닷컴에 필요했던 것은 탄력적인 확장성, 즉 필요할 때 언제든 리소스 인스턴스를 추가로 생성할 수 있는 능력이었습니다. 이제 클라우드 제공업체들은 이러한 기능을 기본적으로 제공하지만, 초기 전자상거래 기업들은 자체 인프라를 관리해야 했고, 많은 기업들이 이전에는 들어본 적 없는 현상, 즉 지나친 성공이 오히려 사업을 망하게 한다는 사실에 희생양이 되었습니다. 펫츠닷컴의 몰락과 이와 유사한 여러 실패 사례들은 소프트웨어 아키텍처를 설계할 때 이러한 위험 요소를 더욱 면밀히 고려해야 한다는 점을 아키텍트들에게 일깨워주었습니다.

아키텍처와 인프라의 교차점은 운영 아키텍처 특성을 구현하는 데 매우 중요합니다. 아키텍처가 높은 확장성을 지원할 수 있다고 해서 실제로 지원된다는 보장은 없습니다. 해당 인프라가 이를 지원하지 않으면 확장성은 불가능합니다(Pets.com 사례 참조). 고객사 현장에서 우리는 아키텍처와 인프라 간의 불일치로 인해 발생한 아키텍처 실패에 대해 아키텍트와 개발자가 비난받는 사례를 너무나 자주 목격해 왔습니다.

대부분의 경우 이러한 불일치는 아키텍트와 인프라 및 운영 담당자 간의 소통과 협업 부족으로 발생합니다. 아키텍트는 종종 인프라가 확장성, 응답성, 내결함성, 성능, 가용성, 탄력성 등과 같은 특성에 미치는 영향을 제대로 인식하지 못합니다. 이러한 불일치가 DevOps라는 분야를 탄생시켰습니다.

오랫동안 많은 기업들은 운영과 소프트웨어 개발을 별개의 것으로 간주하고 비용 절감 차원에서 제3자에게 아웃소싱하는 경우가 많았습니다. 1990년대와 2000년대에는 운영이 아웃소싱되어 아키텍트의 통제 범위를 벗어날 것이라는 가정 하에 방어적인 아키텍처 설계가 많이 이루어졌습니다. (이에 대한 좋은 예는 16 장을 참조하십시오 .) 그러나 2000년대 중반부터 기업들은 운영 관련 여러 요소를 통합한 새로운 형태의 아키텍처를 실험하기 시작했습니다. 예를 들어, 오케스트레이션 기반 SOA와 같은 기존 아키텍처 방식은 이를 지원하기 위해 정교한 도구와 프레임워크를 필요로 했습니다.

확장성 및 탄력성과 같은 기능은 구현을 상당히 복잡하게 만들었습니다.
그래서 아키텍트들은 확장성, 성능, 탄력성 및 기타 여러 기능을 내부적으로 처리할 수 있는 아키텍처를 구축했습니다. 그 결과, 이러한 아키텍처는 훨씬 더 복잡해졌습니다.

마이크로서비스 아키텍처 스타일을 만든 사람들은 운영 관련 문제는 운영팀에서 처리하는 것이 더 효율적이라는 점을 깨달았습니다. 아키텍처와 운영 간의 협력 관계를 구축함으로써, 설계자들은 설계를 간소화하고 운영팀이 가장 잘하는 부분은 운영팀에 맡길 수 있다는 것을 알게 되었습니다. 그들은 운영팀과 협력하여 마이크로서비스를 개발했고, 이는 훗날 데브옵스(DevOps) 운동의 토대가 되었습니다. 데브옵스는 여러모로 도움이 되었지만, 아키텍처와 인프라의 이러한 접점은 여전히 대부분의 기업에게 어려운 과제로 남아 있습니다.

인프라 자체는 큰 문제가 되지 않지만, 클라우드 환경은 아키텍처와 제대로 맞지 않을 수 있습니다. 예를 들어, 서비스를 여러 지역이나 가용성 영역에 배포하면 인메모리 복제 캐시 및 분산 캐시가 제공하는 성능 및 데이터 무결성 이점이 감소하거나 완전히 사라질 수 있습니다. 마찬가지로, 서비스, 컨테이너 또는 Kubernetes Pod를 동일한 가상 머신에 함께 배치하면 성능이 크게 향상되지만 확장성, 내결함성, 가용성 및 탄력성에는 부정적인 영향을 미칩니다.

아키텍처와 인프라를 조화시키려면 아키텍트와 인프라 팀 구성원 간의 긴밀한 소통과 협업, 또는 DevOps 방식을 도입하여 모든 이해관계자가 핵심 운영 문제를 이해해야 합니다. 그래야만 아키텍트가 선택한 아키텍처의 진정한 운영적 이점을 실현할 수 있으며, 바로 이러한 이유로 저희가 별 다섯 개 만점을 드린 것입니다.

아키텍처 및 데이터 토폴로지

아키텍처와 데이터 토폴로지의 상호 작용은 종종 간과됩니다. 잘못된 데이터베이스 유형이나 토폴로지를 선택하면 아키텍처에 악영향을 미치고 최고의 아키텍처적 특성을 무력화할 수 있습니다. 예를 들어, 모놀리식 데이터베이스는 데이터 일관성과 트랜잭션 지원에는 우수하지만 확장성과 내결함성을 저해할 수 있습니다. 마찬가지로, 분산 데이터베이스 토폴로지는 확장성과 변경 관리에는 뛰어나지만 시스템의 데이터 무결성, 데이터 일관성 및 성능을 저하시킬 수 있습니다.

다음 섹션에서는 아키텍처와 데이터 토폴로지의 교차점에 대해 설명합니다.

데이터베이스 토폴로지 15장

에서 논의했듯이 데이터베이스 토폴로지는 아키텍처 내에서 물리적 데이터베이스가 어떻게 구성되는지를 나타냅니다. 기본적인 토폴로지에는 모놀리식 데이터베이스, 분산 도메인 기반 데이터베이스, 또는 서비스별 분산 데이터베이스 토폴로지의 세 가지가 있습니다. 그림 26-4는 이러한 기본적인 데이터베이스 토폴로지 유형을 보여줍니다.

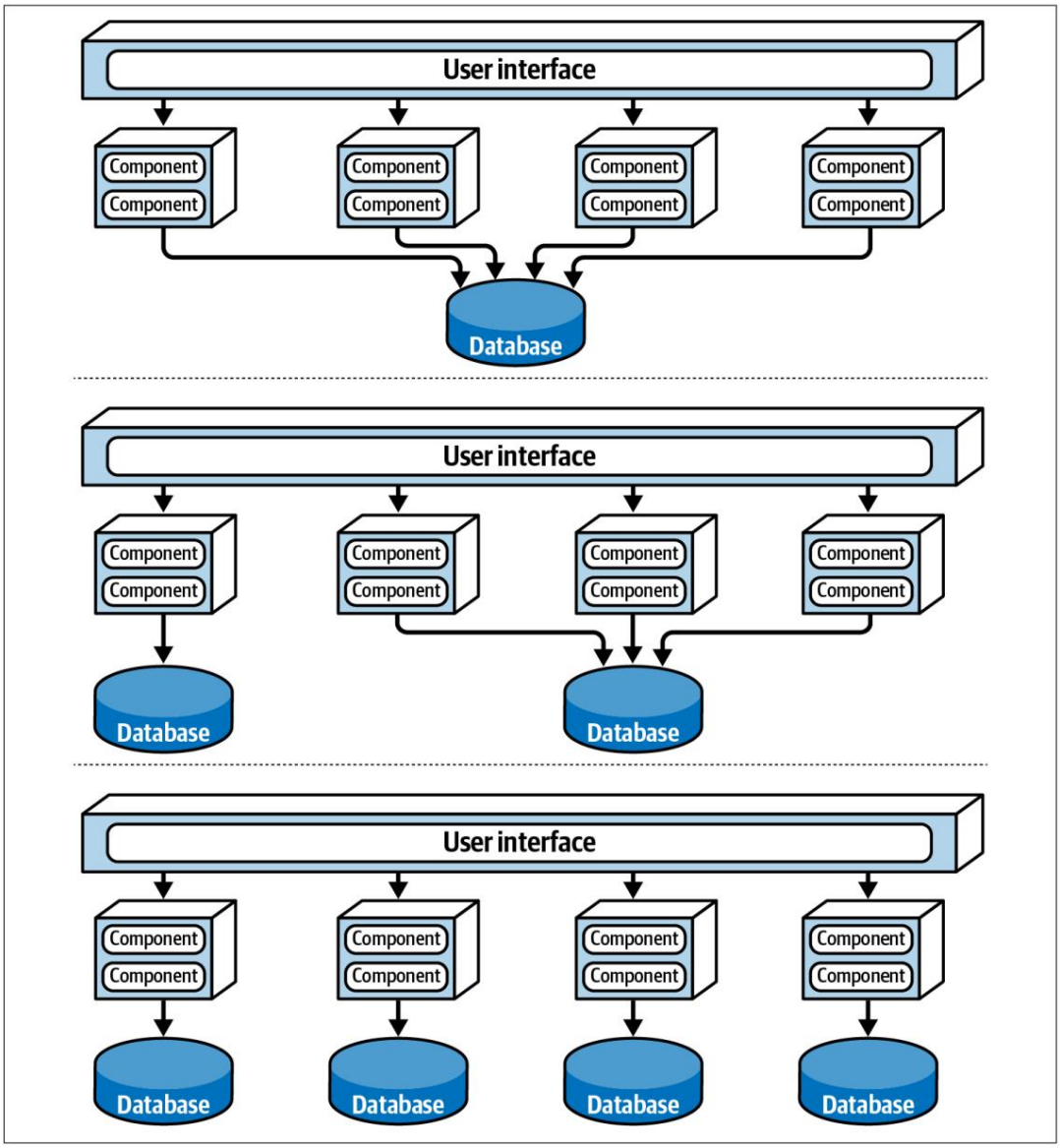

그림 26-4. 일반적인 데이터베이스 토폴로지 유형

데이터베이스 토폴로지는 아키텍처와 일치해야 제대로 작동합니다. 예를 들어, 마이크로서비스 아키텍처는 일반적으로 서비스별로 데이터베이스를 사용합니다.

엄격한 경계 컨텍스트를 유지하기 위한 패턴입니다. 이러한 적절한 정렬이 없으면 아키텍트가 변경 사항을 제어하기가 매우 어려워지고, 내결함성, 확장성, 탄력성, 유지 관리 용이성, 테스트 용이성 및 배포 용이성과 같은 시스템의 운영 특성이 모두 저하될 것입니다. 하지만 서비스 기반 아키텍처(14 장 참조)와 같은 일부 스타일은 물리적 데이터베이스 토폴로지 측면에서 더 유연합니다.

건축적 특징

이 책 2부 에서 우리는 모든 아키텍처 스타일에는 강점(4~5점 만점)과 약점(1~2점)이 있다는 것을 보여드렸습니다. 데이터베이스 유형도 마찬가지이며, 시스템의 아키텍처 강점과 그에 상응하는 데이터베이스 유형의 강점을 조화시키는 것이 중요합니다. 저희 저서 『소프트웨어 아키텍처: 어려운 부분들』에서 저자들은 관계형, 키-값, 문서, 컬럼형, 그래프, NoSQL 등 6가지 데이터베이스 유형의 특성을 평가했습니다. 확장성과 탄력성은 마이크로서비스, 이벤트 기반, 공간 기반 아키텍처의 강점이라는 것을 기억하실 겁니다. 이러한 강점은 키-값 및 컬럼형 데이터베이스에도 해당되므로, 이러한 데이터베이스 유형은 해당 아키텍처 특성을 강화하는 데 좋은 선택입니다.

데이터 구조

저장되고 접근되는 데이터의 구조 또한 이러한 교집합에 영향을 미치는 요소입니다.
데이터 구조가 관계형, 즉 상호 의존적인 관계의 계층 구조를 기반으로 하는 경우 관계형 데이터베이스가 적합합니다. 그러나 키-값 쌍을 관계형 데이터베이스에 저장하는 것은 데이터베이스와 아키텍처 모두에서 비효율성을 초래할 수 있는 부적절한 접근 방식입니다. 모든 데이터가 동일한 구조를 갖는 것은 아니므로 이 점을 유의해야 합니다. 어떤 데이터는 관계형일 수 있고, 어떤 데이터는 문서 기반일 수 있으며(특히 JSON 기반 이벤트 또는 요청 페이로드를 저장할 때), 또 다른 데이터는 키-값 기반일 수 있습니다. 주어진 아키텍처 내에서 데이터 구조의 다양성을 고려하여 가능한 한 다중 언어 데이터베이스를 활용하는 것이 좋습니다.

읽기/쓰기 우선순위는 비즈니스

문제가 높은 읽기 또는 쓰기 볼륨과 관련된 경우, 데이터베이스 토폴로지를 아키텍처에 맞추는 데 중요한 정보입니다. 예를 들어, 아키텍처에서 쓰기 볼륨을 높은 빈도로 처리해야 하고 읽기 빈도는 낮다면 컬럼형 데이터베이스가 적합합니다. 반대로 읽기 볼륨이 높은 것이 우선순위라면 키-값 데이터베이스, 문서 데이터베이스 또는 그래프 데이터베이스가 더 적합합니다. 시스템에서 읽기와 쓰기의 우선순위가 비슷하다면 관계형 데이터베이스와 NewSQL 데이터베이스가 좋은 선택이 될 수 있습니다. 이 요소를 제대로 고려하지 않으면 시스템 성능이 저하될 수 있습니다.

건축 및 엔지니어링 실무

20세기 후반, 워터폴 방식과 스크럼, 익스트림 프로그래밍, 린, 크리스털 등 다양한 애자일 방식을 포함하여 수십 가지의 소프트웨어 개발 방법론이 인기를 얻었습니다. 당시 대부분의 아키텍트는 이러한 방법론들이 소프트웨어 아키텍처에 영향을 미치지 않는다고 생각했으며, 개발을 완전히 별개의 프로세스로 간주했습니다.
하지만 최근 몇 년 동안 엔지니어링 기술의 발전으로 소프트웨어 아키텍처에 프로세스 관련 문제가 더욱 중요해졌습니다. 소프트웨어 개발 프로세스와 엔지니어링 관행을 구분하는 것이 유용합니다. 여기서 프로세스란 팀 구성 및 관리 방식, 회의 진행 방식, 워크플로 구성 방식 등 사람들이 조직되고 상호 작용하는 방식을 의미합니다. 반면 엔지니어링 관행은 팀이 소프트웨어를 개발하고 배포하는 데 사용하는 프로세스에 구애받지 않는 기술과 도구를 가리킵니다. 예를 들어, 익스트림 프로그래밍(XP), 지속적 통합(CI), 지속적 배포(CD), 테스트 주도 개발(TDD)은 모두 특정 프로세스에 의존하지 않는 검증된 엔지니어링 관행입니다. 따라서 소프트웨어 엔지니어링이라는 용어는 소프트웨어 개발과 이러한 관행을 모두 포괄합니다.

엔지니어링 실무에 집중하는 것이 중요합니다. 소프트웨어 개발은 다른 성숙한 엔지니어링 분야가 가진 여러 특징을 갖추지 못하고 있습니다. 예를 들어, 토목 엔지니어는 소프트웨어 엔지니어보다 구조물의 변화를 훨씬 더 정확하게 예측할 수 있습니다. 즉, 소프트웨어 개발의 가장 큰 약점은 추정, 즉 얼마나 많은 시간, 얼마나 많은 자원, 얼마나 많은 비용이 필요한지 예측하는 것입니다. 이러한 어려움의 원인 중 하나는 전통적인 추정 방식이 소프트웨어 개발의 탐색적 특성과 개발 과정에서 흔히 발생하는 불확실성을 고려하지 못한다는 점입니다.

프로세스는 아키텍처와 대체로 분리되어 있지만, 반복적인 프로세스가 그 본질에 더 잘 맞습니다. 워터폴과 같은 구식 프로세스를 사용하여 마이크로서비스와 같은 최신 시스템을 구축하려고 하면 상당한 마찰이 발생할 것입니다. 애자일 방법론이 특히 빛을 발하는 아키텍처 측면 중 하나는 한 아키텍처 스타일에서 다른 스타일로 마이그레이션하는 것입니다. 애자일 방법론은 피드백 루프가 긴밀하고 스트 랭글러 패턴 이나 기능 토글과 같은 기법을 장려하기 때문에 계획 위주의 프로세스보다 이러한 변경을 더 효과적으로 지원합니다.

아키텍트는 종종 프로젝트의 기술 리더 역할도 수행하는데, 이는 팀이 사용할 엔지니어링 방식을 결정해야 한다는 것을 의미합니다. 아키텍처를 선택하기 전에 문제 영역을 신중하게 고려하는 것처럼, 아키텍트는 자신이 선택한 아키텍처 스타일과 엔지니어링 방식이 조화를 이루도록 해야 합니다. 예를 들어, 마이크로서비스 아키텍처 철학은 팀이 머신 프로비저닝, 테스트, 배포와 같은 작업을 자동화할 것이라는 전제를 깔고 있습니다. 구식 운영 부서, 수동 프로세스, 그리고 부족한 테스트로 마이크로서비스 아키텍처를 구축하려 한다면 실패할 가능성이 높습니다. 서로 다른 문제 영역에 적합한 아키텍처 스타일이 있듯이, 서로 다른 엔지니어링 방식 또한 마찬가지입니다.

익스트림 프로그래밍부터 지속적 배포에 이르기까지, 엔지니어링 기술의 발전으로 새로운 아키텍처 기능이 가능해짐에 따라 사고의 진화는 계속되고 있습니다. 닐의 저서 『진화적 아키텍처 구축(Building Evolutionary Architectures)』은 엔지니어링 기술과 아키텍처의 교차점에 대한 새로운 사고방식을 제시하여 아키텍처 거버넌스 자동화 방식을 개선하는 데 도움을 줍니다. 이 책은 아키텍처 특성에 대한 중요한 새로운 용어와 사고방식을 제공하고, 시간이 지남에 따라 자연스럽게 변화하는 아키텍처를 구축하는 기법을 다룹니다.

소프트웨어 개발 세계에서는 그 무엇도 정적인 것이 아닙니다. 아키텍트는 특정 기준을 충족하는 시스템을 설계할 수 있지만, 구현 과정과 불가피한 변화 속에서도 설계가 유효하게 유지되도록 하려면 진화하는 아키텍처가 필요합니다.

진화적 아키텍처 구축(Building Evolutionary Architectures)에서는 시간이 지남에 따라 발생하는 변화에 대응하여 아키텍처 특성을 보호(및 관리)하기 위해 아키텍처 적합성 함수를 사용하는 개념을 소개합니다. 6장 에서 살펴본 바와 같이, 아키텍처 적합성 함수는 특정 아키텍처 특성의 객관적인 무결성 평가를 수행하는 방법입니다. 이러한 평가는 메트릭, 단위 테스트, 모니터링, 카오스 엔지니어링 등 다양한 메커니즘을 포함할 수 있습니다.

적합성 함수를 활용하여 이러한 교차점을 어떻게 조율할 수 있는지 살펴보려면, 빠른 시장 출시가 필요한 비즈니스 문제를 생각해 보겠습니다. 시장 출시 시간은 민첩성, 즉 시스템이 변화에 신속하게 대응하는 능력과 직결됩니다. 민첩성은 유지보수성, 테스트 용이성, 배포 용이성으로 구성된 복합적인 아키텍처 특성입니다(6 장 참조). 이 세 가지 아키텍처 특성은 모두 엔지니어링 관행과 절차의 영향을 받으며, 따라서 적합성 함수를 통해 측정하고 추적할 수 있습니다. 예를 들어, 마이크로서비스 아키텍처와 서비스 기반 아키텍처는 모두 높은 수준의 민첩성을 지원합니다. 그러나 이러한 아키텍처 특성을 둘러싼 엔지니어링 관행이 아키텍처와 일치하지 않으면 시스템은 민첩성 목표와 요구 사항을 충족하지 못합니다. 적합성 함수는 불일치를 식별하는 데 도움을 주어 아키텍트가 엔지니어링 관행을 아키텍처에 맞추도록(또는 그 반대로) 조치를 취하도록 유도할 수 있습니다.

아키텍처 및 팀 토폴로지

이 책 2부 에서 논의했듯이 , 팀 구성은 소프트웨어 아키텍처에 직접적인 영향을 미칠 수 있으며, 그 반대의 경우도 마찬가지입니다. 이러한 연관성은 매우 중요하기 때문에 이 책에서 소개하는 각 아키텍처 스타일에 대해 팀 구성에 대한 섹션을 포함했습니다.

팀 토폴로지를 아키텍처와 연계하는 가장 기본적인 방법 중 하나는 파티셔닝 유형을 활용하는 것입니다. 아키텍처와 마찬가지로 팀도 도메인별로 분할하거나 기술적별로 분할할 수 있습니다. 도메인별로 분할된 팀은 도메인 영역별로 구성되며, 일반적으로 팀 전체에 걸쳐 전문성을 갖춘 다기능 팀입니다. 예를 들어, 특정 도메인 영역별 팀은 다음과 같이 구성됩니다.

도메인 분할 팀은 시스템의 고객 접점 부분에 집중할 수 있으며, 따라서 UI부터 데이터베이스에 이르기까지 고객 관련 기능의 엔드투엔드 처리를 담당할 수 있습니다. 반면 기술 분할 팀은 아키텍처의 특정 기술 기능에 각각 집중하며 일반적으로 기술 범주별로 구성됩니다. 예를 들어 UI 팀, 백엔드 처리 팀, 공유 서비스 팀, 데이터베이스 팀은 계층형 아키텍처 스타일에 매우 잘 부합합니다. 또는 이러한 팀을 비즈니스 기능 팀과 데이터 동기화 팀으로 기술적으로 분할할 수도 있으며, 이는 공간 기반 아키텍처 스타일에 잘 맞습니다.

팀 구성 방식을 이해하는 것은 시스템의 성공을 보장하는 데 매우 중요합니다. 조직의 팀 구성이 아키텍처와 일치하지 않으면 팀은 아키텍처를 구현하고 유지 관리하는 데 어려움을 겪게 되고, 결국 비즈니스 목표를 달성하지 못할 가능성이 높습니다.

아키텍처 및 시스템 통합

시스템은 드물게 완전히 독립적으로 존재합니다. 대부분의 시스템은 다른 시스템으로부터 추가적인 처리와 데이터를 필요로 하며, 바로 이 지점에서 아키텍처와 시스템 통합이 중요해집니다. 한 시스템이 추가 처리를 수행하거나 데이터를 검색하기 위해 다른 시스템과 통신해야 할 때, 시스템 아키텍트는 여러 가지 문제와 고려 사항에 직면하게 됩니다. 예를 들어, 호출받는 시스템은 사용 가능한 상태일까요? 호출하는 시스템의 요구 사항을 충족할 만큼 확장성과 성능이 충분할까요?

아키텍트가 시스템 통합에 충분히 집중하지 않으면 시스템 간의 정적 및 동적 결합으로 인해 확장성이 떨어지고 반응성이 부족하며 민첩성이 결여된 아키텍처가 만들어지는 경우가 많습니다. 다른 시스템과 통합할 때는 사용할 통신 프로토콜, 시스템 간 계약 유형, 시스템 아키텍처 특성의 호환성, 그리고 통합 과정에서 각 시스템의 아키텍처적 특성(quantum)이 유지되는지 여부를 고려해야 합니다.

건축과 기업

모든 기업에는 일련의 표준과 지침 원칙이 있습니다. 여기서 기업이란 회사(또는 회사 내 부서나 사업부) 내의 모든 시스템과 제품의 집합체를 의미합니다. 예를 들어, 많은 기업은 시스템 유형에 관계없이 아키텍처 솔루션에 특정 보안 표준, 관행 또는 절차를 적용합니다. 기업 표준에는 플랫폼, 기술, 문서화 표준, 다이어그램 표준 등이 포함될 수 있습니다. 기업 수준의 표준과 관행을 숙지하고 아키텍처가 이에 부합하도록 설계되었는지 확인해야 합니다.

우리는 설계자가 기업 차원의 관행, 표준 및 절차를 무시하는 상황을 여러 번 경험했습니다. 그 결과는 대개 설계자가 잘못된 방향으로 나아가는 것이었습니다.

아무리 기술적으로 효과적인 솔루션이라도 실패한 "일회성" 솔루션으로 간주되어 폐기되는 경우가 있습니다. 성공을 보장하기 위해서는 아키텍처를 기업 관행에 맞춰 설계하는 것이 얼마나 중요한지 아무리 강조해도 지나치지 않습니다.

건축과 비즈니스 환경

비즈니스 환경은 시스템 아키텍처에 중대한 직접적인 영향을 미치며(역으로도 마찬가지), 비즈니스 환경은 끊임없이 변화합니다. 회사가 생존을 위해 심각한 비용 절감 조치를 취하고 있는지, 아니면 공격적으로 확장하고 있는지에 따라 상황이 달라집니다. 변동성이 크고 경쟁이 치열한 시장에서 틈새시장을 찾기 위해 매 분기마다 사업 방향을 전환하고 재포지셔닝을 하고 있는지, 아니면 안정적인 위치에 있는지도 중요합니다. 유능한 소프트웨어 아키텍트는 회사의 현재 상황과 방향을 이해하고, 핵심 시스템의 아키텍처를 비즈니스 환경에 맞춰 설계합니다.

우리는 이러한 정렬을 도메인-아키텍처 동형성이라고 부릅니다. 예를 들어, 극단적인 비용 절감 조치를 시행하는 기업은 구축 및 유지 관리에 비용이 많이 드는 마이크로서비스 또는 공간 기반 아키텍처와 잘 맞지 않을 것입니다. 반대로, 인수 합병을 통해 공격적으로 확장하는 기업은 진화 및 적응력이 부족한 모놀리식 아키텍처 스타일에는 적합하지 않을 것입니다.

건축가들이 이러한 교차점에서 흔히 직면하는 문제 중 하나는 비즈니스 변화, 특히 예측 불가능한 변화입니다. 도널드 럼스펠드 전 미국 국방장관은 다음과 같이 유명한 말을 남겼습니다 .

> 우리가 알고 있는 것들이 있습니다. 즉, 우리가 알고 있다고 확신하는 것들이죠. 또한 우리가 알지 못하는 것들이 있다는 것을 알고 있는 것들도 있습니다. 다시 말해, 우리가 알지 못한다는 사실조차 모르는 것들 말입니다. 하지만 우리가 알지 못한다는 사실조차 모르는 것들도 있습니다.

많은 제품과 시스템은 알려진 미지수 목록, 즉 개발자가 도메인과 기술에 대해 알아야 할 변화할 것들을 가지고 시작합니다. 그러나 이러한 시스템들은 또한 알려지지 않은 미지수, 즉 아무도 예상하지 못했지만 예기치 않게 나타나는 것들의 희생양이 되기도 합니다. "알 수 없는 미지수"는 소프트웨어 시스템의 적입니다. 이것이 바로 "사전 설계 중심"의 소프트웨어 개발 프로젝트가 어려움을 겪는 이유입니다. 아키텍트는 알려지지 않은 미지수를 고려하여 설계할 수 없기 때문입니다. 마크의 말을 인용하자면,

> 모든 아키텍처는 예측 불가능한 요소 때문에 반복적인 과정을 거치게 됩니다. 애자일은 단지 이러한 점을 인지하고 더 빨리 실행에 옮길 뿐입니다.

소프트웨어 아키텍처 변경을 계획하는 것은 어렵습니다. 진화적 아키텍처 방식은 반복적 아키텍처 방식과 마찬가지로 끊임없이 변화하는 비즈니스 환경에 대응하는 데 도움이 됩니다.
이식성, 확장성, 진화 가능성 및 적응성과 같은 아키텍처적 특성을 수용하는 것은 소프트웨어 아키텍처를 더욱 유연하고 변화에 잘 적응할 수 있도록 만드는 데 도움이 됩니다.

복잡성 이론과 소프트웨어 설계를 전문으로 하는 경험 많은 소프트웨어 아키텍트인 배리 오라일리는 끊임 없는 비즈니스 변화에 대한 새로운 사고방식인 잔여성 이론을 제시했습니다. 그의 저서 『잔여물: 소프트웨어 아키텍처의 시간, 변화, 불확실성』(Leanpub, 2024)에서 오라일리는 비즈니스 변화를 스트레스 요인으로, 그에 따른 아키텍처 변화를 잔여물로 취급하는 기법을 설명합니다.

그의 이론은 건축가가 변화에 대응하여 아키텍처에 점점 더 많은 잔여 요소를 적용함에 따라, 이러한 잔여 요소들이 결국 건축가가 예측할 수 없는 미지의 변화에 대응하게 되면서, 복잡성 이론의 임계점에 도달한 아키텍처를 만들어낸다는 것입니다. 흥미로운 이론이며, 저희는 이를 예의주시하고 있습니다.

건축과 생성형 인공지능

2025년 초, 이 책의 2판을 집필하는 시점에서 생성형 인공지능(Gen AI)과 대규모 언어 모델(LLM)은 소프트웨어 개발 및 설계 분야에 깊숙이 자리 잡고 있습니다. 많은 기업들이 이전에는 사람이 수작업으로만 수행했던 작업을 자동화하기 위해 이러한 기술을 시스템에 통합하고 있습니다. 놀랍게도 Gen AI는 소프트웨어 아키텍처와도 밀접한 관련이 있습니다. 아키텍트들은 소프트웨어 아키텍처에 LLM을 통합하고 있으며, 일부는 어려운 문제를 해결하는 데 도움을 받기 위해 Gen AI 도구를 활용하기도 합니다.

생성형 AI를 아키텍처에 통합하는 한 가지 방법으로 추상화와 모듈성을 활용하는 것을 권장

합니다. 하나의 LLM을 다른 LLM으로 신속하게 교체할 수 있어야 하며, 다양한 LLM에서 생성된 결과에 대한 가이드라인(rails)과 평가 기준(evals)을 설정할 수 있어야 합니다.

예를 들어, 구직 회사가 편견을 줄이고 구직자의 인구통계학적 정보나 기타 요인보다는 기술에 초점을 맞추기 위해 Gen AI를 활용하여 이력서를 익명화한다고 가정해 보겠습니다. 이 작업은 일반적으로 사람이 수행하지만, LLM(Learning Level Modeling)을 사용하면 쉽게 처리할 수 있습니다. 하지만 LLM의 결과는 정확할까요? 이력서에서 너무 많은 정보를 삭제하지는 않을까요? 인구통계학적 정보는 너무 많이 남겨두지는 않을까요? 이러한 시스템에서는 샘플과 지표를 수집하고 LLM 엔진을 비교할 수 있는 것이 매우 중요합니다. Langfuse 와 같은 도구는 아키텍처 내에서 이러한 관찰 가능성을 구축하는 데 도움이 됩니다.

건축가 보조 도구로서의 생성형 AI

적절한 프롬프트가 주어지면 Copilot 과 같은 LLM은 개발자의 시간과 노력을 크게 절약해주는 소스 코드를 생성할 수 있습니다. "중복되는 숫자가 없는 고유한 4자리 PIN 번호를 생성하는 C# 프로그래밍 언어의 소스 코드를 작성하세요"와 같이 매우 구체적이고 확정적인 문제를 해결하는 데 탁월합니다. 하지만

LLM 기술은 소프트웨어 아키텍트의 일반적인 작업에 어떻게 도움이 될까요? 다음은 아키텍처 관련 질문의 일반적인 예입니다.

- 위험 평가: "이 아키텍처 내에 위험 영역이 있습니까?"
- 위험 완화: "이 위험을 어떻게 해결해야 할까요?" • 안티패턴: "이 아키텍처에 흔히 나타나는 안티패턴이 있나요?" • 의사 결정: "이 워크플로에 오케스트레이션을 사용해야 할까요, 아니면 코레오그래피를 사용해야 할까요?"

이 두 번째 개정판을 집필하는 시점(2025년 초) 기준으로, 우리는 이러한 시도에서 큰 성공을 거두지 못했습니다. LLM(Limited Leadership Development Manager)에게 특정 상황에 마이크로서비스 아키텍처와 공간 기반 아키텍처 중 어느 것이 가장 적합한지 물어보면 정답을 얻는 경우는 거의 없습니다. 왜 그럴까요? 이 책에서 설명했듯이 소프트웨어 아키텍처의 모든 것은 장단점이 존재하기 때문입니다. LLM은 지식을 이해하는 데는 탁월하지만, 오늘날까지도 적절한 결정을 내리는 데 필요한 지혜가 부족합니다. 이러한 지혜에는 방대한 맥락이 포함되어 있기 때문에, 아키텍트가 직접 비즈니스 문제를 해결하는 것이 LLM에게 문제와 그 확장된 환경 및 맥락에 대해 모두 가르치는 것보다 훨씬 빠릅니다. 우리가 고려해야 할 여덟 가지 다른 교차점을 제시했다는 사실만으로도 이것이 얼마나 어려운 과제인지 충분히 알 수 있습니다.

그렇긴 하지만, 유망한 도구들이 몇 가지 눈에 띕니다. 예를 들어, Thoughtworks Haiven은 아키텍처 다이어그램을 해석하고 소프트웨어 아키텍처를 완벽하게 설명할 수 있어, 다이어그램을 XML과 같은 기계 판독 가능한 형식으로 내보내고 이를 LLM에 입력하는 번거로움을 덜어줍니다. 사용자는 해당 정보를 가져온 후 Haiven에게 아키텍처에 대한 간단한 질문, 예를 들어 병목 현상이나 문제점을 식별할 수 있는지 등을 물어볼 수 있습니다. 다른 연구 사례로는 LLM을 사용하여 PlantUML 다이어그램이나 아키텍처에 대한 의사 언어 설명을 실행 가능한 ArchUnit 코드로 변환하여 시스템 구조를 관리하는 방식이 있습니다. 이 분야에서 많은 연구가 진행되고 있으므로, 향후 몇 년 동안 인공지능이 아키텍트를 지원하는 방식에 빠른 변화가 있을 것으로 예상됩니다.

요약하자면, 소프

트웨어 아키텍처는 조직의 여러 측면을 아우르는 총체적인 활동입니다. 유능한 소프트웨어 아키텍트는 아키텍처를 설계하고 유지하는 것이 단순히 특정 아키텍처 스타일을 선택하고 구현을 진행하는 것 이상이라는 점을 인식해야 합니다. 아키텍처가 환경의 다른 측면들과 조화를 이루도록 하고, 3부 에서 설명한 의사소통 및 협업 기술을 활용하여 이러한 조화를 실현하는 것 또한 중요합니다.