제21장

건축적 결정

건축가의 핵심적인 역할 중 하나는 건축적인 결정을 내리는 것입니다.
아키텍처 결정은 일반적으로 애플리케이션이나 시스템의 구조와 관련되지만, 특히 기술적 결정이 아키텍처적 특성에 영향을 미치는 경우에는 기술적 결정도 포함될 수 있습니다. 어떤 맥락에서든, 좋은 아키텍처 결정이란 개발팀이 올바른 기술적 선택을 하도록 안내하는 것입니다. 아키텍처 결정을 내리려면 충분한 관련 정보를 수집하고, 결정의 타당성을 입증하고, 문서화하고, 적절한 이해관계자에게 효과적으로 전달해야 합니다.

아키텍처 결정 안티패턴

프로그래머 앤드류 코닉은 안티패턴을 처음 시작할 때는 좋은 아이디어처럼 보이지만 결국 문제를 야기하는 것이라고 정의합니다. 또 다른 정의로는 부정적인 결과를 초래하는 반복적인 프로세스가 있습니다. 아키텍트가 아키텍처 결정을 내릴 때 흔히 발생하는 세 가지 안티패턴은 자산 보호 안티패턴, 반복적인 패턴, 그리고 이메일 중심 아키텍처 안티패턴입니다. 이 세 가지 안티패턴은 일반적으로 점진적인 흐름을 따릅니다. 자산 보호 안티패턴을 극복하면 반복적인 패턴으로 이어지고, 이를 극복하면 이메일 중심 아키텍처 안티패턴으로 이어집니다. 효과적이고 정확한 아키텍처 결정을 내리려면 이 세 가지 안티패턴을 모두 극복해야 합니다.

자산 보호 안티패턴 은 아키텍트가 잘못된 선택을

할까 봐 두려워 아키텍처 관련 결정을 미루거나 회피할 때 발생합니다. 이를 극복하는 방법은 두 가지가 있습니다. 첫 번째는 중요한 아키텍처 결정을 내릴 때, 즉 충분한 정보가 확보된 마지막 순간까지 기다리는 것입니다.

결정을 정당화하고 타당성을 검증하는 데는 시간이 걸리지만, 개발팀의 업무를 지연시키거나 아키텍트가 분석 마비(Analysis Paralysis)라는 악순환에 빠져 영원히 분석에만 매달리는 상황이 발생하지 않도록 해야 합니다. 결정을 미루는 데 드는 비용이 결정을 내리는 데 따른 위험을 초과하는 시점을 파악하는 것이 중요합니다. 그림 21-1 에서 볼 수 있듯이, 의사결정 시간 척도의 초기 단계에서는 결정에 소요되는 시간이 적기 때문에 비용(실선)은 낮지만, 문제나 해결책에 대한 정보가 부족하기 때문에 위험(점선)은 높습니다. 결정을 미루는 시간이 길어질수록 비용은 증가하지만, 아키텍트가 문제와 가능한 대안에 대해 더 완벽하게 분석할 수 있으므로 위험은 감소합니다. 결정을 내리는 시점은 이 두 요소가 교차하고 비용 증가가 위험 감소를 초과하는 지점입니다.

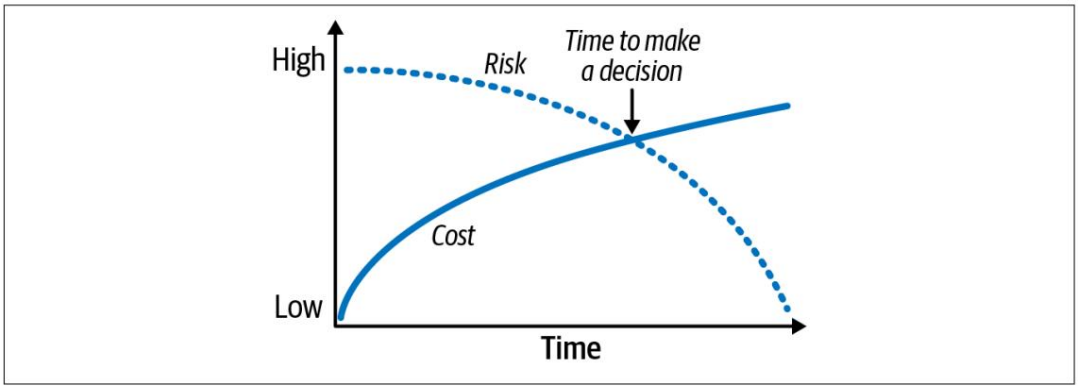

그림 21-1. 마지막 책임 순간

이러한 안티패턴을 피하는 또 다른 방법은 개발팀과 협력하여 결정 사항이 예상대로 구현될 수 있도록 하는 것입니다. 이는 매우 중요한데, 어떤 아키텍트도 특정 기술과 관련된 모든 문제에 대한 모든 세부 사항을 알 수는 없기 때문입니다. 개발팀과 긴밀히 협력함으로써 아키텍트는 신속하게 대응하고, 더 많은 통찰력을 얻고, 잘못된 결정을 내릴 위험을 줄일 수 있습니다.

이 점을 설명하기 위해, 아키텍트인 당신이 제품 관련 참조 데이터(제품 설명, 무게, 크기 등)를 해당 정보가 필요한 모든 서비스 인스턴스에 캐시하기로 결정했다고 가정해 보겠습니다. 당신은 카탈로그 로그 서비스가 소유하는 기본 캐시를 사용하는 읽기 전용 복제 캐시를 통해 이를 구현하기로 결정했습니다. (복제 캐시 또는 인메모리 캐시란 제품 정보가 변경되거나 새 제품이 추가될 경우 카탈로그 서비스가 캐시를 업데이트하고, 이 업데이트된 정보가 복제 캐시 제품을 통해 해당 데이터가 필요한 다른 모든 서비스로 복제되는 것을 의미합니다.) 이러한 결정의 근거는 서비스 간의 결합도를 낮추고 서비스 간 호출 없이 데이터를 효율적으로 공유하는 것입니다. 그러나 이러한 아키텍처 결정을 구현하는 개발 팀은 일부 서비스의 확장성 요구 사항 때문에 이 결정이 더 많은 것을 필요로 한다는 것을 알게 됩니다.

프로세스 내 메모리 사용량이 사용 가능한 메모리보다 많습니다. 이러한 팀들과 긴밀하게 협업하기 때문에 문제를 신속하게 파악하고 그에 따라 아키텍처 설계를 조정할 수 있습니다.

'그라운드호그 데이' 안티패턴은 건축가

가 특정 결정을 내린 이유를 알지 못해 계속해서 같은 문제를 논의하고 또 논의하지만, 최종적인 해결책이나 합의에 도달하지 못하는 상황을 말합니다. 이 패턴은 1993년 영화 '그라운드호그 데이'에서 빌 머레이가 연기한 주인공이 매일 2월 2일을 반복해서 겪어야 하는 설정에서 유래했습니다.

이러한 안티패턴은 아키텍트가 자신의 결정에 대한 정당성을 제시하지 못하거나 (혹은 완전히 제시하지 못하기 때문에) 발생합니다. 아키텍처 결정에 대해서는 기술적 정당성과 비즈니스적 정당성 모두를 제시하는 것이 중요합니다.

예를 들어, 모놀리식 애플리케이션을 여러 개의 개별 서비스로 분리하기로 결정했다고 가정해 보겠습니다. 분리의 근거는 애플리케이션의 기능적 측면을 분리하여 각 부분이 가상 머신 리소스를 덜 사용하고 별도로 유지 관리 및 배포할 수 있도록 하는 것입니다. 이는 기술적으로 타당한 근거이지만, 비즈니스적인 근거, 즉 기업이 이러한 아키텍처 재구성에 비용을 지불해야 하는 이유가 빠져 있습니다. 이러한 결정에 대한 타당한 비즈니스적 근거로는 새로운 비즈니스 기능을 더 빠르게 제공하여 출시 기간을 단축하는 것을 들 수 있습니다. 또 다른 근거로는 새로운 기능 개발 및 출시와 관련된 비용을 절감하는 것을 들 수 있습니다.

아키텍처 설계를 정당화할 때 비즈니스 가치를 제공하는 것은 매우 중요합니다. 또한, 해당 설계가 애초에 타당한지 여부를 판단하는 좋은 기준이 되기도 합니다. 만약 비즈니스 가치를 제공하지 못한다면, 설계자는 해당 설계를 재고해야 할 것입니다.

가장 일반적인 사업적 타당성 근거 네 가지는 비용, 출시 기간, 사용자 만족도, 그리고 전략적 포지셔닝입니다. 사업 이해관계자들에게 무엇이 중요한지 고려해야 합니다. 이해관계자들이 출시 기간을 더 중요하게 생각한다면, 비용 절감만을 근거로 특정 결정을 정당화하는 것은 적절하지 않을 수 있습니다.

이메일 중심 아키텍처 안티패턴은 아키텍트가 결정을 내

리고 그 타당성을 충분히 입증한 후에도 종종 나타나는 현상입니다. 이는 사람들이 아키텍처 관련 결정을 잃어버리거나 잊어버리거나, 심지어 그런 결정이 내려졌다는 사실조차 몰라서 구현할 수 없는 경우에 발생합니다. 이 안티패턴을 극복하는 핵심은 아키텍처 관련 결정을 효과적으로 소통하는 것입니다. 이메일은 훌륭한 소통 도구이지만, 문서 저장 시스템으로는 적합하지 않습니다.

다행히 건축가는 아키텍처 결정 사항을 효과적으로 전달하는 방법을 배우면 이메일 중심 아키텍처라는 안티패턴을 쉽게 피할 수 있습니다. 첫째, 이메일 본문에 결정 사항을 직접 포함하지 않도록 주의해야 합니다. 이렇게 하면 각 이메일에 결정 사항 사본이 포함되어 여러 시스템에 기록되기 때문입니다. 또한, 이러한 이메일 중 상당수는 결정에 대한 중요한 세부 정보(정당화 근거 포함)를 누락하여 '반복되는 상황'을 초래합니다. 게다가, 해당 아키텍처 결정이 변경되거나 대체될 경우, 관련자들이 모두 수정된 결정 사항을 받았는지 확인하기 어렵습니다.

더 나은 접근 방식은 이메일 본문에 결정의 성격과 맥락만 언급하고, 아키텍처 결정 및 관련 세부 정보가 저장된 단일 기록 시스템(위키 페이지 링크 또는 파일 시스템의 문서 참조)에 대한 링크를 제공하는 것입니다.

건축 설계 결정과 관련된 다음 이메일을 살펴보세요.

> "안녕하세요, 산드라님. 서비스 간 통신과 관련하여 귀하에게 직접적인 영향을 미치는 중요한 결정을 내렸습니다. 아래 링크를 통해 해당 결정을 확인해 주세요…"

첫 번째 문장의 시작 부분을 보면 맥락(서비스 간 통신)은 언급되지만 실제 결정 자체는 언급되지 않았다는 점에 주목하세요. 첫 번째 문장의 두 번째 부분도 중요합니다. 아키텍처 결정이 특정 사용자에게 직접적인 영향을 미치지 않는다면, 굳이 그 사용자에게 알릴 필요가 있을까요? 이는 어떤 이해관계자(개발자 포함)에게 아키텍처 결정을 직접 알려야 하는지 판단하는 데 유용한 기준이 됩니다. 이 예시의 두 번째 문장은 아키텍처 결정이 한 곳에 기록되도록 링크를 제공하여, 모든 결정에 대한 단일 기록 시스템을 구축합니다.

건축적 중요성

많은 건축가들은 특정 기술과 관련된 결정은 건축적인 결정이 아니라 기술적인 결정이라고 생각합니다. 하지만 이것이 항상 맞는 말은 아닙니다.

만약 건축가가 특정 아키텍처 특성(예: 성능 또는 확장성)을 직접적으로 지원하기 때문에 특정 기술을 사용하기로 결정했다면, 그것 또한 아키텍처적 결정입니다.

유명한 소프트웨어 아키텍트이자 『릴리스 잇!』(Release It!) 2판(Pragmatic Bookshelf, 2018)의 저자인 마이클 니가드는 아키텍트가 책임져야 할 결정(그리고 아키텍처적 결정이란 무엇인가)에 대한 문제를 다루면서 '아키텍처적으로 중요한 결정'이라는 용어를 만들었습니다. 니가드에 따르면, 아키텍처적으로 중요한 결정이란 시스템의 구조, 비기능적 특성, 의존성, 인터페이스 또는 구축 기법에 영향을 미치는 결정을 말합니다.

이 맥락에서 구조란 사용되는 아키텍처 패턴이나 스타일에 영향을 미치는 결정을 의미합니다. 예를 들어, 아키텍트가 여러 코드 그룹 간에 코드를 공유하기로 결정하는 것이 구조에 해당합니다.

마이크로서비스의 경계 컨텍스트에 영향을 미치고, 결과적으로 시스템 구조에 영향을 미칩니다.

시스템의 비기능적 특성은 개발 또는 유지 관리되는 시스템에 중요한 아키텍처적 특성을 의미합니다. 예를 들어, 기술 선택이 성능에 영향을 미치고 성능이 애플리케이션의 중요한 측면이라면, 특정 제품, 프레임워크 또는 기술을 명시하더라도 해당 선택은 아키텍처적 결정이 됩니다.

의존성은 시스템 내 구성 요소 및/또는 서비스 간의 연결 지점을 의미합니다. 의존성은 확장성, 모듈성, 민첩성, 테스트 용이성, 신뢰성 등과 같은 아키텍처 특성에 영향을 미칠 수 있으므로 의존성에 대한 결정은 아키텍처 관련 결정이 됩니다.

인터페이스는 서비스와 구성 요소에 접근하고 이를 구성하는 방식을 나타냅니다. 일반적으로 게이트웨이, 통합 허브, 서비스 버스, 어댑터 또는 API 프록시를 통해 이루어집니다. 인터페이스에 대한 결정을 내릴 때는 버전 관리 및 사용 중단 전략을 포함한 계약을 정의하는 것이 중요합니다. 인터페이스는 시스템을 사용하는 다른 사용자에게 영향을 미치므로 아키텍처적으로 매우 중요합니다.

마지막으로, 구축 기술은 플랫폼, 프레임워크, 도구, 심지어 프로세스에 대한 결정을 의미하며, 이러한 결정은 기술적인 성격을 띠지만 아키텍처의 특정 측면에 영향을 미칠 수 있습니다.

건축 결정 기록

아키텍처 관련 의사결정을 문서화하는 가장 효과적인 방법 중 하나는 아키텍처 의사결정 기록 (ADR)을 활용하는 것입니다. 마이클 니가드는 2011년 블로그 게시물에서 ADR의 중요성을 처음으로 강조했으며, 2017년에는 Thoughtworks Technology Radar에서 이 기법의 광범위한 도입을 권장했습니다.

ADR(아키텍처 설계 보고서)은 특정 아키텍처 설계 결정을 설명하는 짧은 텍스트 파일(일반적으로 1~2페이지 분량)입니다. ADR은 일반 텍스트나 위키 페이지 템플릿을 사용하여 작성할 수 있지만, 대개 AsciiDoc 이나 Markdown과 같은 텍스트 문서 형식으로 작성됩니다.

ADR 관리를 위한 도구도 있습니다. 『테스트를 통한 객체 지향 소프트웨어 개발(Growing Object-Oriented Software, Guided by Tests)』(Addison-Wesley, 2009)의 공동 저자인 Nat Pryce는 ADR Tools 라는 오픈 소스 도구를 개발했는데, 이 도구는 번호 체계, 위치, 대체된 로직 등을 포함한 ADR 관리를 위한 명령줄 인터페이스를 제공합니다. 독일의 소프트웨어 엔지니어인 Micha Kops는 아키텍처 결정 기록을 관리하기 위해 ADR 도구를 사용하는 훌륭한 예시들을 제공합니다.

기본 구조

ADR(대체 분쟁 해결)의 기본 구조는 제목, 상태, 배경, 결정, 결과의 다섯 가지 주요 섹션으로 구성됩니다. 일반적으로 기본 구조에 규정 준수 및 참고 사항 섹션을 추가합니다. 규정 준수 섹션은 아키텍처 결정이 어떻게 관리되고 시행될지(수동 또는 자동화된 적합성 기능을 통해)를 고려하고 문서화하는 공간입니다. 참고 사항 섹션은 결정 작성자, 승인자, 작성 시점 등과 같은 결정 관련 메타데이터를 포함하는 공간입니다.

(그림 21-2 에 나와 있는) 이 기본 구조를 확장하여 필요한 다른 섹션을 추가해도 괜찮습니다 . 단, 템플릿은 일관성 있고 간결하게 유지해야 합니다. 좋은 예로는 다른 가능한 해결책들을 분석하는 '대안' 섹션을 추가하는 것이 있습니다.

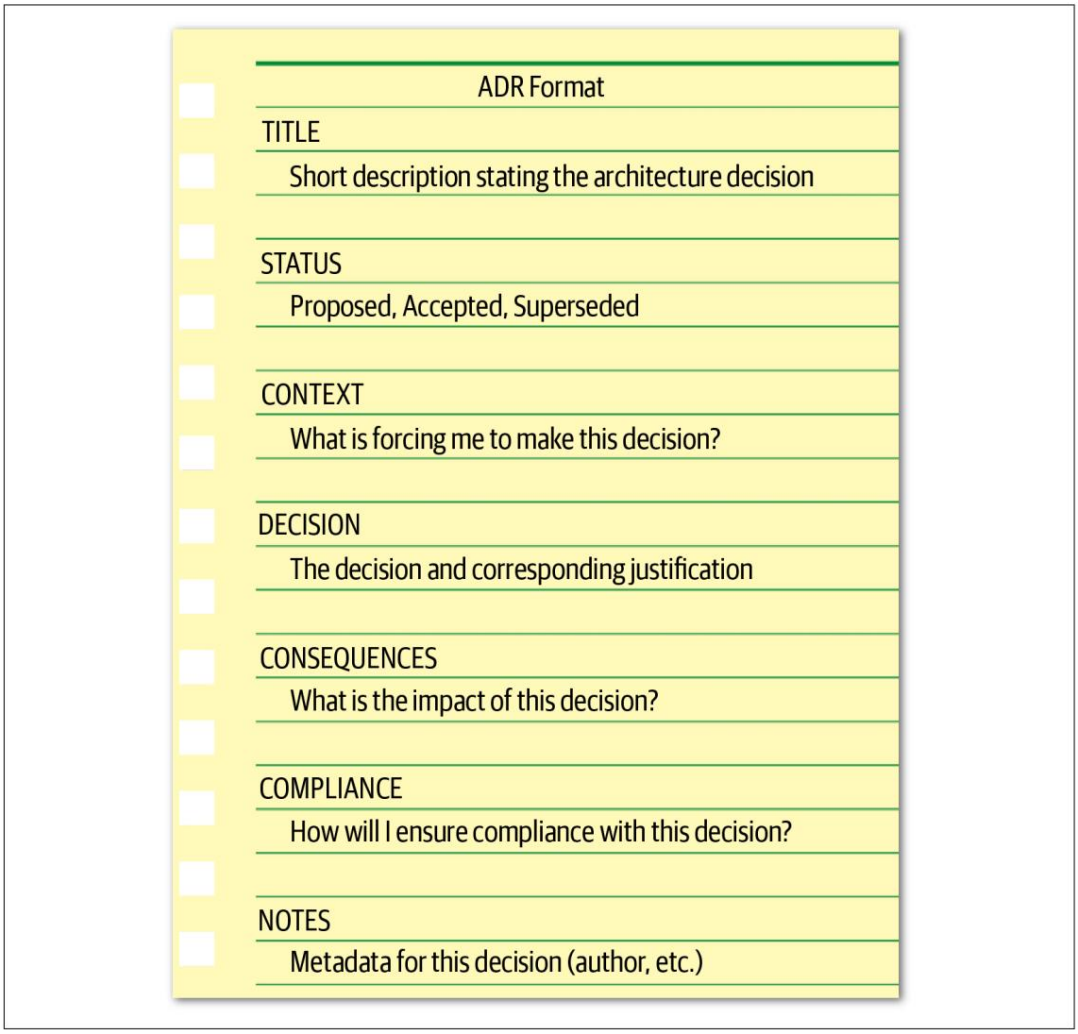

그림 21-2. 기본 ADR 구조

제목

ADR 제목은 일반적으로 순차적으로 번호가 매겨지며 아키텍처 결정에 대한 간략한 설명 문구를 포함합니다. 예를 들어, 주문 서비스와 결제 서비스 간에 비동기 메시징을 사용하기로 한 결정을 설명하는 ADR 제목은 "42. 주문 및 결제 서비스 간 비동기 메시징 사용"과 같을 수 있습니다.

제목은 짧고 간결해야 하지만, 결정의 성격과 맥락에 대한 모호함을 없애기 위해 충분히 설명적이어야 합니다.

상태

모든 ADR은 제안됨, 승인됨, 또는 대체됨의 세 가지 상태 중 하나를 가집니다. 제안됨 상태는 상위 의사 결정권자 또는 아키텍처 관리 기구(예: 아키텍처 검토 위원회)의 승인을 받아야 함을 의미합니다.

'승인됨'은 결정이 승인되어 시행 준비가 완료되었음을 의미합니다.
대체되었다는 것은 해당 결정이 변경되어 다른 대체 분쟁 해결 절차(ADR)에 의해 대체되었음을 의미합니다.
대체됨(Superseded) 상태는 이전 ADR의 상태가 승인됨(Accepted)이었음을 전제로 합니다. 즉, 제안된 ADR은 다른 ADR에 의해 대체되지 않고, 승인될 때까지 수정될 뿐입니다.

대체됨 상태는 어떤 결정이 내려졌는지, 당시 왜 그런 결정이 내려졌는지, 새로운 결정은 무엇인지, 그리고 왜 변경되었는지에 대한 기록을 유지하는 강력한 방법입니다. 일반적으로 대체된 분쟁 해결(ADR) 결정에는 이를 대체한 결정의 번호가 표시됩니다. 마찬가지로, 다른 ADR을 대체하는 결정에는 그 결정이 대체한 ADR의 번호가 표시됩니다.

예를 들어, ADR 42("주문 및 결제 서비스 간 비동기 메시징 사용")가 승인 상태라고 가정해 보겠습니다. 이후 결제 서비스의 구현 및 위치가 변경되어 두 서비스 간에 REST API를 사용하기로 결정했습니다. 따라서 이러한 변경 사항을 문서화하기 위해 새로운 ADR(번호 68)을 생성합니다. 이에 따른 상태는 다음과 같습니다.

> ADR 42. 주문 및 결제 서비스 간 비동기 메시징 사용 상태: 68로 대체됨 ADR 68. 주문
> 및 결제 서비스 간 REST 사
> 용 상태: 승인됨, 42를 대체함

ADR 42와 68 간의 연결 및 이력 추적을 통해 ADR 68과 관련하여 필연적으로 제기되는 "메시징을 사용하는 것은 어떻습니까?"라는 질문을 피할 수 있습니다.

대체 분쟁 해결(ADR) 및 의견 요청(RFC)

ADR 초안이나 의견을 배포하는 것은 아키텍트가 더 많은 이해관계자들과 함께 자신의 가정과 주장을 검증하는 데 도움이 될 수 있습니다. 개발자의 참여를 유도하고 협업을 시작하는 효과적인 방법은 RFC(Request for Comments)라는 새로운 상태 유형을 만들고 검토자가 피드백을 완료할 마감일을 지정하는 것입니다. 마감일이 되면 아키텍트는 의견을 분석하고 필요한 조정을 거쳐 최종 결정을 내린 후 상태를 제안됨(또는 아키텍트에게 승인 권한이 있는 경우 승인됨)으로 설정할 수 있습니다.

RFC 상태를 포함하는 ADR은 다음과 같습니다.

상태
의견 수렴 요청, 마감일: 2026년 1월 9일

ADR의 상태 섹션에서 또 다른 중요한 측면은 아키텍트와 그의 상사 또는 수석 아키텍트가 아키텍처 결정 승인 기준에 대해 논의하고 아키텍트가 단독으로 승인할 수 있는지, 아니면 상위 아키텍트, 아키텍처 검토 위원회 또는 기타 관리 기관의 승인을 받아야 하는지를 결정하도록 한다는 점입니다.

이러한 논의를 시작하기에 좋은 세 가지 요소는 비용, 팀 간 영향, 그리고 보안입니다. 비용에는 소프트웨어 구매 또는 라이선스 비용, 추가 하드웨어 비용, 그리고 아키텍처 설계를 구현하는 데 필요한 전반적인 노력 수준이 포함되어야 합니다. 이를 추정하려면 아키텍처 설계 구현에 필요한 예상 시간을 회사의 정규직 환산 인력(FTE)으로 곱하면 됩니다. 프로젝트 소유자 또는 프로젝트 관리자가 일반적으로 FTE 수치를 알고 있습니다. 이러한 논의를 통해 예를 들어 아키텍처 설계 비용이 특정 금액을 초과하는 경우, 해당 설계를 '제안' 상태로 설정하고 다른 담당자의 승인을 받아야 한다는 데 모두가 동의할 수 있습니다. 아키텍처 설계가 다른 팀이나 시스템에 영향을 미치거나 보안상의 문제가 있는 경우에는 상위 관리자 또는 수석 아키텍트의 승인을 받아야 합니다.

팀이 기준과 그에 따른 제한(예: "5,000달러를 초과하는 비용은 아키텍처 검토 위원회의 승인을 받아야 함")을 정하고 합의하면, 모든 아키텍트가 ADR을 작성할 때 자신의 아키텍처 결정을 언제 승인할 수 있고 언제 승인할 수 없는지 알 수 있도록 해당 내용을 잘 문서화해야 합니다.

문맥

ADR의 맥락(Context) 섹션에서는 작용하는 요인들을 명시합니다. 다시 말해, "어떤 상황이 내가 이러한 결정을 내리도록 강요하는가?"를 설명합니다. ADR의 이 섹션을 통해 아키텍트는 구체적인 상황을 설명하고 가능한 대안들을 간결하게 제시할 수 있습니다. 아키텍트가 각 분석에 대한 내용을 문서화해야 하는 경우,

대안을 자세히 설명하려면, 해당 분석을 맥락 섹션에 포함하는 대신 대안 섹션을 추가하십시오.

'맥락' 섹션은 건축물의 특정 영역을 기록할 수 있는 공간도 제공합니다. 건축가는 맥락을 설명함으로써 건축물 자체를 설명하는 것입니다.
이전 섹션의 예시를 사용하면 컨텍스트 섹션은 다음과 같이 작성될 수 있습니다. "주문 서비스는 현재 진행 중인 주문에 대한 결제를 위해 결제 서비스로 정보를 전달해야 합니다. 이는 REST 또는 비동기 메시징을 사용하여 수행할 수 있습니다."
이 간결한 진술은 시나리오뿐만 아니라 고려된 대안까지 명시하고 있다는 점에 주목하십시오.

결정

ADR의 결정 섹션에는 아키텍처 결정에 대한 설명과 함께 충분한 근거가 포함됩니다. Nygard는 아키텍처 결정을 수동적인 어조보다는 긍정적이고 단호한 어조로 표현할 것을 권장합니다. 예를 들어, 서비스 간 비동기 메시징을 사용하기로 한 결정은 "우리는 서비스 간 비동기 메시징을 사용할 것입니다."라고 표현하는 것이 좋습니다. 이는 "저는 서비스 간 비동기 메시징이 최선의 선택이라고 생각합니다."라고 표현하는 것보다 훨씬 효과적입니다. 후자는 결정의 내용이나 결정이 내려졌는지조차 명확하지 않고, 단지 아키텍트의 의견일 뿐입니다.

ADR(대체 분쟁 해결)의 결정 섹션에서 가장 강력한 측면 중 하나는 아키텍트가 결정에 대한 정당성을 강조할 수 있도록 한다는 점입니다. 어떤 것이 어떻게 작동하는지를 이해하는 것보다 왜 그런 결정이 내려졌는지 이해하는 것이 훨씬 중요합니다. 이는 개발자와 다른 이해관계자들이 결정의 근거를 더 잘 이해하고, 결과적으로 그 결정에 동의할 가능성을 높여줍니다.

이 점을 설명하기 위해, 매우 높은 응답성이 요구되는 두 특정 서비스 간의 통신에 네트워크 지연 시간을 줄이기 위해 Google의 원격 프로시저 호출 (gRPC) 을 사용 하기로 결정했다고 가정해 보겠습니다. 몇 년 후, 팀의 새로운 아키텍트가 서비스 간 통신의 일관성을 높이기 위해 gRPC 대신 REST를 사용하기로 결정합니다. 이 새로운 아키텍트는 애초에 gRPC를 선택한 이유를 이해하지 못했기 때문에, 그의 결정은 결국 지연 시간에 상당한 영향을 미쳐 상위 시스템에서 타임아웃을 발생시킵니다. 만약 이 새로운 아키텍트가 ADR(아키텍트 문제 해결사)에게 접근할 수 있었다면, gRPC를 사용하기로 한 원래 결정이 (긴밀한 서비스 결합이라는 대가를 치르더라도) 지연 시간을 줄이기 위한 것이었다는 점을 이해하고 이러한 문제를 예방할 수 있었을 것입니다.

결과 모든 건축

가의 결정은 좋든 나쁘든 어떤 식으로든 영향을 미칩니다. ADR의 결과 섹션은 건축가가 건축적 결정의 전반적인 영향을 설명하도록 요구하며, 이를 통해 부정적인 영향이 긍정적인 영향보다 큰지 여부를 판단할 수 있도록 합니다.

이 섹션은 의사 결정 과정에서 수행한 트레이드오프 분석을 기록하기에도 좋은 곳입니다. 예를 들어, 웹사이트에 리뷰를 게시하기 위해 비동기(전송 후 처리) 메시징을 사용하기로 결정했다고 가정해 보겠습니다. 이 결정의 근거는 응답성을 개선하는 것(3,100밀리초에서 25밀리초로 단축)입니다. 사용자는 실제 리뷰가 게시될 때까지 기다릴 필요 없이 메시지가 큐에 전송되는 시간만 기다리면 되기 때문입니다. 하지만 개발팀의 한 구성원은 비동기 요청과 관련된 오류 처리의 복잡성 때문에 이것이 좋은 생각이 아니라고 주장합니다. "만약 누군가 욕설이 포함된 리뷰를 게시하면 어떻게 하죠?" 이 팀 구성원이 모르는 사실은, 여러분이 비즈니스 이해관계자 및 다른 아키텍트들과 함께 이 결정의 트레이드오프를 분석할 때 바로 그 문제를 논의했고, 대기 시간을 늘리고 리뷰 게시 성공 여부에 대한 피드백을 제공하는 것보다 응답성을 개선하고 복잡한 오류 처리를 하는 것이 더 낫다는 결론을 내렸다는 것입니다. 만약 이 결정이 대체 분쟁 해결 절차(ADR)를 사용하여 문서화되었다면, 결과 분석 부분에 이러한 절충안 분석을 포함시켜 이와 같은 의견 불일치를 방지할 수 있었을 것입니다.

규정 준수

규정 준수 섹션은 ADR의 표준 섹션은 아니지만, 추가하는 것을 강력히 권장합니다. 규정 준수 섹션에서는 아키텍처 결정에 대한 평가 및 관리 방법을 명시합니다. 해당 결정에 대한 규정 준수 검사는 수동으로 수행될 것인지, 아니면 적합성 함수를 사용하여 자동화될 수 있는지를 설명해야 합니다. 자동화가 가능하다면, 아키텍트는 적합성 함수를 작성하는 방법과 해당 아키텍처 결정의 규정 준수 여부를 측정하는 데 필요한 코드베이스 변경 사항을 구체적으로 명시할 수 있습니다.

예를 들어, 기존의 n계층 구조 아키텍처(그림 21-3 참조)에서 비즈니스 계층의 비즈니스 객체가 사용하는 모든 공유 객체가 공유 기능을 격리하고 포함하기 위해 공유 서비스 계층에 있어야 한다는 결정을 내렸다고 가정해 보겠습니다.

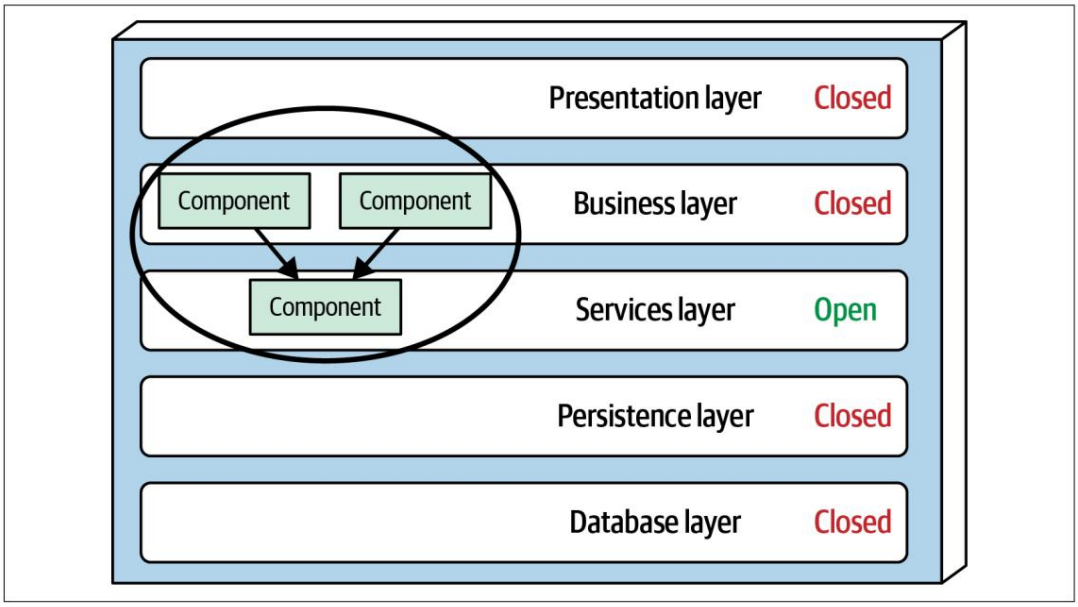

그림 21-3. 건축 설계 결정의 예

이러한 아키텍처 결정은 Java의 ArchUnit 및 C#의 NetArchTest를 비롯한 다양한 자동화 도구를 사용하여 측정하고 관리할 수 있습니다 . Java용 ArchUnit을 사용한 이러한 아키텍처 결정에 대한 자동화된 적합성 함수 테스트는 다음과 같습니다.

```
@Test
public void shared_services_should_reside_in_services_layer()
        { classes().that().areAnnotatedWith(SharedService.class)
                .should().resideInAPackage("..services..") .check(myClasses);

}
```

이 자동화된 적합성 기능을 구현하려면 Java 어노테이션 (@SharedService) 을 생성 하고 이를 모든 공유 클래스에 추가하여 이러한 관리 방식을 지원하는 새로운 스토리를 작성해야 합니다.

메모

표준 ADR에는 포함되지 않지만 추가하는 것을 강력히 권장하는 또 다른 섹션은 메모 섹션입니다. 이 섹션에는 ADR에 대한 다양한 메타데이터가 포함됩니다.

- 원저자
- 승인일
- 승인자
- 대체일
- 최종 수정일
- 수정자:
- 최종 수정일

Git과 같은 버전 관리 시스템에 ADR을 저장하는 경우에도 저장소에서 지원하는 것 이상의 추가 메타데이터를 저장하는 것이 유용합니다. 아키텍트가 ADR을 저장하는 방식이나 위치와 관계없이 이 섹션을 추가하는 것이 좋습니다.

예시 로 제시

된 Going, Going, Gone(GGG) 경매 시스템에는 수십 가지의 아키텍처 설계 결정이 포함되어 있습니다. 입찰자와 경매 진행자 사용자 인터페이스를 분리하고, 이벤트 기반 방식과 마이크로서비스를 결합한 하이브리드 아키텍처를 사용하며, 실시간 전송 프로토콜(RTP)을 활용하여 영상을 캡처하고, 단일 API 게이트웨이를 사용하고, 메시지 전송을 위한 별도의 큐를 사용하는 것 등은 아키텍트가 내리는 아키텍처 설계 결정의 일부에 불과합니다. 아키텍트가 내리는 모든 아키텍처 설계 결정은 아무리 당연해 보이더라도 문서화하고 그 타당성을 입증해야 합니다.

그림 21-4는 GGG 경매 시스템 내의 아키텍처 결정 중 하나를 보여줍니다. 즉, 단일 발행-구독 토픽(또는 REST조차도) 대신 입찰 캡처, 입찰 스트리머 및 입찰 추적 서비스 간에 별도의 지점 간 큐를 사용하는 것입니다.

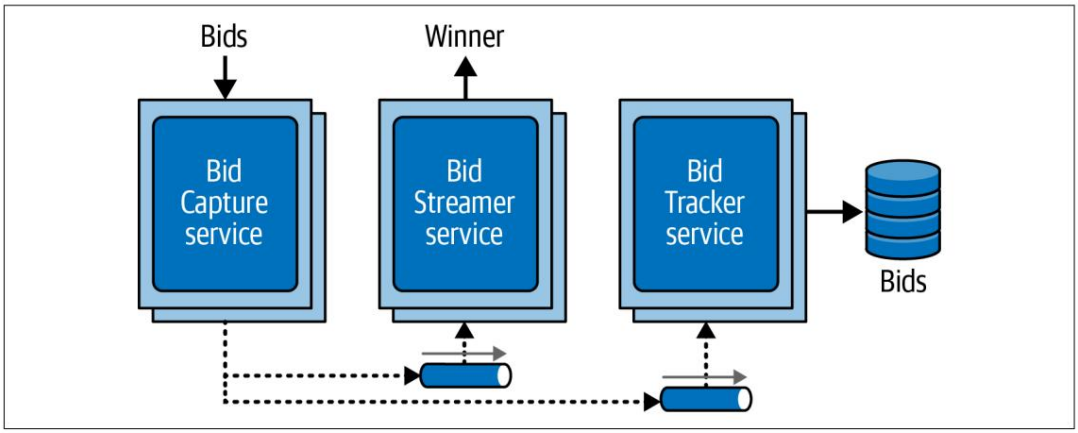

그림 21-4. 서비스 간 pub/sub 사용

이 결정을 정당화할 수 있는 대체 분쟁 해결(ADR) 절차가 없다면, 이 시스템의 설계 및 개발에 관련된 다른 사람들이 동의하지 않고 다른 방식으로 시스템을 구현하기로 결정할 수도 있습니다.

다음은 이러한 아키텍처 결정에 대한 ADR(대체 분쟁 해결)의 예입니다.

ADR 76. 입찰 스트리머 및 입찰자 추적 서비스에 대한 별도의 대기열

상태

수락됨

문맥

입찰 접수 서비스는 입찰을 수신하면 해당 입찰을 입찰 스트리머 서비스와 입찰자 추적 서비스로 전달해야 합니다. 이는 단일 토픽(발행/구독), 각 서비스별 별도 큐(지점 간 연결), 또는 온라인 경매 API 계층을 통한 REST API 방식을 사용하여 구현할 수 있습니다.

결정: 입찰 스트리

머 와 입찰자 추적 서비스 에 각각 별도의 대기열을 사용하겠습니다 .

입찰 캡처 서비스는 입찰 스트리머 서비스나 입찰자 추적 서비스 로부터 어떠한 정보도 필요로 하지 않습니다 (통신은 단방향입니다).

입찰 스트리머 서비스는 입찰 캡처 서비스 에서 수락된 정확한 순서대로 입찰을 수신해야 합니다 . 메시징 및 큐를 사용하면 선입선출(FIFO) 큐를 활용하여 스트림의 입찰 순서를 자동으로 보장할 수 있습니다.

같은 금액으로 여러 입찰이 들어오는 경우 (예: "100달러인가요?").
Bid Streamer 서비스는 해당 금액에 대해 수신된 첫 번째 입찰만 필요로 하는 반면, Bidder Tracker는 수신된 모든 입찰이 필요합니다. 토픽(pub/sub) 방식을 사용하면 Bid Streamer는 이전 금액과 동일한 입찰을 무시해야 하므로, Bid Streamer는 인스턴스 간에 공유 상태를 저장해야 합니다.

Bid Streamer 서비스는 상품에 대한 입찰가를 메모리 캐시에 저장하는 반면, Bidder Tracker는 입찰가를 데이터베이스에 저장합니다. 따라서 Bidder Tracker 는
속도가 느려지고 백프레셔가 필요할 수 있습니다. 전용 입찰자 추적 큐를 사용하면 이러한 전용 백프레셔 지점을 제공할 수 있습니다.

결과적으로 메시지 큐의 클러

스터링과 고가용성이 필요하게 될 것입니다.

이 결정으로 인해 입찰 수주 서비스는 동일한 정보를 여러 대기열에 전송해야 합니다.

내부 입찰 이벤트는 API 계층에서 수행되는 보안 검사를 우회합니다.
업데이트: 2025년 1월 14일 ARB 회의에서 검토 결과, ARB는 이것이 수용 가능한 절충안이며 이러한 서비스 간 입찰 행사에 추가적인 보안 검사가 필요하지 않다고 결정했습니다.

규정 준수를 위해 입찰 캡

처 서비스, 입찰 스트리머 서비스 및 입찰자 추적 서비스 간에 비동기 게시/구독 메시지가 사용되고 있는지 확인하기 위해 정기적인 수동 코드 검토를 실시할 것입니다 .

참고 사항
저자: 수바시니 나델라

승인: ARB 회의 위원, 2025년 1월 14일

최종 업데이트: 2025년 1월 14일

ADR 저장하기 아

키텍트가 ADR을 생성하면 어딘가에 저장해야 합니다. 저장 위치는 중요하지 않으며, 각 아키텍처 결정은 별도의 파일이나 위키 페이지로 관리되어야 합니다. 일부 아키텍트는 소스 코드와 동일한 Git 저장소에 ADR을 저장하여 소스 코드처럼 버전 관리 및 추적을 할 수 있도록 합니다.

하지만 규모가 큰 조직의 경우, 몇 가지 이유로 이러한 관행을 권장하지 않습니다.
첫째, 아키텍처 결정 사항을 확인해야 하는 모든 사람이 ADR이 저장된 Git 저장소에 접근 권한이 있는 것은 아닙니다. 둘째, 애플리케이션의 Git 저장소는 통합 아키텍처 결정, 엔터프라이즈 아키텍처 결정 또는 모든 애플리케이션에 공통적인 결정과 같이 외부 컨텍스트를 갖는 ADR을 저장하기에 적합하지 않습니다. 이러한 이유로 모든 사람이 접근할 수 있는 전용 ADR Git 저장소, 위키(위키 템플릿 사용), 또는 위키나 기타 문서 렌더링 소프트웨어에서 쉽게 접근할 수 있는 공유 파일 서버의 공유 디렉터리에 ADR을 저장하는 것을 권장합니다.

그림 21-5는 이러한 디렉터리 구조(또는 위키 페이지 탐색 구조)가 어떻게 보일 수 있는지를 보여줍니다.

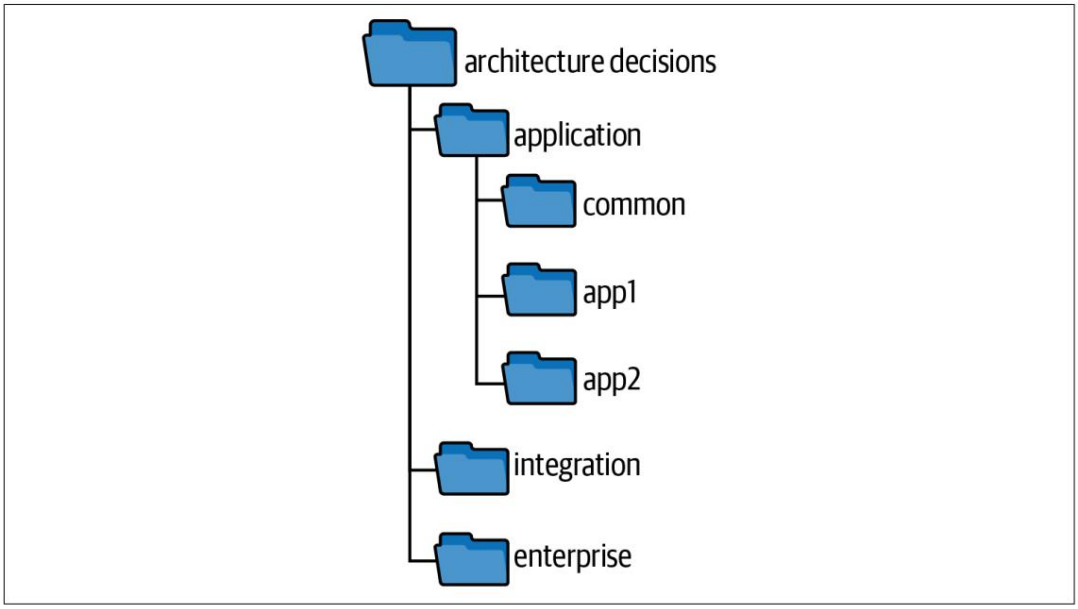

그림 21-5. ADR 저장용 디렉토리 구조 예시

애플리케이션 디렉터리에는 특정 애플리케이션(또는 제품) 컨텍스트에 특화된 아키텍처 결정 사항이 포함되어 있습니다. 이 디렉터리는 다시 여러 하위 디렉터리로 나뉩니다.

흔한

공통 하위 디렉터리는 "모든 프레임워크 관련 클래스에는 해당 클래스가 기본 프레임워크 코드에 속함을 나타내는 어노테이션(@Java의 경우 Framework) 또는 속성([Framework] C#의 경우)이 포함됩니다."와 같이 모든 애플리케이션에 적용되는 아키텍처 결정을 위한 것입니다.

application 디

렉터리 아래의 하위 디렉터리는 특정 애플리케이션 또는 시스템 컨텍스트에 해당하며 해당 애플리케이션 또는 시스템에 특정한 아키텍처 결정 사항을 포함합니다(이 예에서는 app1 및 app2 애플리케이션).

통합 디렉터리에

는 애플리케이션, 시스템 또는 서비스 간의 통신과 관련된 ADR이 포함되어 있습니다.

엔터프라이

즈 아키텍처 ADR(Advanced Database Release)은 엔터프라이즈 디렉터리에 포함되어 있으며, 이는 모든 시스템 및 애플리케이션에 영향을 미치는 전역적인 아키텍처 결정임을 나타냅니다. 엔터프라이즈 아키텍처 ADR의 예로는 "시스템 데이터베이스에 대한 모든 접근은 소유 시스템에서만 허용"하는 것이 있으며, 이는 여러 시스템에서 데이터베이스를 공유하는 것을 방지합니다.

위키에 ADR을 저장할 때도 동일한 구조가 적용되며, 각 디렉터리 구조는 탐색 랜딩 페이지를 나타냅니다. 각 ADR은 각 탐색 랜딩 페이지(애플리케이션, 통합 또는 엔터프라이즈) 내에서 단일 위키 페이지로 표현됩니다.

이 섹션에 제시된 디렉토리 및 랜딩 페이지 이름은 권장 사항 및 예시일 뿐입니다. 회사 상황에 맞는 이름을 자유롭게 선택하세요. 단, 모든 팀에서 일관성 있게 사용해야 합니다.

ADR을 문서화하는 방법

소프트웨어 아키텍처를 문서화하는 것은 항상 어려운 일이었습니다. 아키텍처 다이어그램 작성을 위한 몇 가지 표준(예: 소프트웨어 아키텍트 사이먼 브라운의 C4 모델 또는 오픈 그룹의 ArchiMate 표준)이 등장하고 있지만, 소프트웨어 아키텍처 문서화에 대한 합의된 표준은 존재하지 않습니다. 바로 이 지점에서 ADR이 등장합니다.

ADR(아키텍처 설계 보고서)은 소프트웨어 아키텍처를 문서화하는 효과적인 수단이 될 수 있습니다. 컨텍스트 섹션은 아키텍처적 결정이 필요한 시스템의 특정 영역과 대안을 설명할 수 있는 훌륭한 기회를 제공합니다. 더욱 중요한 것은, 결정 섹션에서 특정 결정을 내리는 이유를 설명한다는 점인데, 이는 아키텍처 문서화의 가장 좋은 형태라고 할 수 있습니다. 결과 섹션에서는 결정에 대한 절충 분석을 설명함으로써 마지막 퍼즐 조각을 완성합니다. 예를 들어, 확장성보다 성능을 선택하는 이유(및 절충점)를 제시할 수 있습니다.

표준에 대한 ADR 활용하기 개발자

들 중 표준을 좋아하는 사람은 거의 없습니다. 안타깝게도 표준은 유용한 목적을 제공하기보다는 통제에 치중하는 경우가 많습니다. ADR을 표준에 활용하면 이러한 잘못된 관행을 바꿀 수 있습니다. 예를 들어, ADR의 '맥락' 섹션에서는 조직이 특정 표준을 채택해야 하는 상황을 설명합니다. 따라서 ADR의 '결정' 섹션에서는 표준이 무엇인지뿐만 아니라, 더 중요하게는 왜 표준이 필요한지까지 명확히 제시할 수 있습니다.

이는 특정 표준이 애초에 존재해야 하는지 여부를 판단하는 데 매우 효과적인 방법입니다. 아키텍트가 그 존재 이유를 정당화할 수 없다면, 그 표준을 설정하고 시행하는 것은 바람직하지 않을 수 있습니다. 또한 개발자들이 특정 표준이 존재하는 이유를 더 잘 이해할수록, 그 표준을 따를 가능성이 높아지고 (결과적으로 이의를 제기하지 않을 가능성도 높아집니다). ADR(대체 분쟁 해결)의 결과 (Consequences) 섹션 역시 표준의 타당성을 판단하는 데 유용한 부분입니다. 이 섹션에서는 아키텍트가 표준의 의미와 결과, 그리고 해당 표준을 구현해야 하는지 여부에 대해 심사숙고하고 문서화해야 합니다.

기존 시스템에서 ADR 사용

많은 아키텍트들은 기존 시스템에 대한 ADR(아키텍트 설계 문서)의 유용성에 의문을 제기합니다. 이미 아키텍처 설계는 완료되었고 시스템은 이미 운영 중인데, 이 시점에서 ADR이 실질적인 의미가 있을까 하는 것입니다. 하지만 ADR은 분명히 의미가 있습니다. ADR은 단순한 문서화 이상의 역할을 합니다. 아키텍트와 개발자가 특정 결정이 내려진 이유와 그 결정이 적절했는지 여부를 이해하는 데 도움을 줍니다.

먼저, 중요한 아키텍처 결정에 대한 ADR(아키텍처 문제 해결 요청)을 작성하고 해당 결정이 올바른지 의문을 제기해 보세요. 예를 들어, 여러 서비스가 하나의 데이터베이스를 공유하고 있다고 가정해 보겠습니다. 왜 그럴까요? 타당한 이유가 있을까요? 데이터를 분리해야 할까요?

기존 시스템에 대안적 의사결정(ADR)을 통합하는 과정의 일부는 이러한 '왜'라는 질문에 대한 해답을 찾기 위한 조사 작업입니다. 안타깝게도 최초 결정을 내린 담당자가 이미 오래전에 회사를 떠났을 경우, 아무도 정확한 답을 알지 못할 수 있습니다. 이러한 경우, 아키텍트가 대안들을 파악하고 각 옵션의 장단점을 분석하여 기존 결정의 타당성을 검증(또는 반증)해야 합니다. 어떤 경우든, 이러한 중요한 결정에 대한 ADR을 작성하는 것은 시스템에 대한 정당성과 근거(그리고 전문가 집단)를 구축하는 데 도움이 되며, 아키텍처의 비효율성과 잘못된 시스템 설계를 식별하는 데에도 기여할 수 있습니다.

생성형 AI와 LLM을 활용한 아키텍처 설계 생성형 AI의 흥미로운 측면 중 하나는 설

계자가 이를 활용하여 의사 결정을 내리고 검증할 수 있는지 여부입니다. 서비스는 하위 시스템으로 데이터를 전송할 때 메시징, 스트리밍 또는 이벤트 소싱 중 어떤 방식을 사용해야 할까요? 데이터베이스는 단일 형태로 유지해야 할까요?

단일 구조로 유지해야 할까요, 아니면 도메인별 데이터베이스로 분할해야 할까요? 결제 처리는 단일 서비스로 배포해야 할까요, 아니면 결제 유형별로 여러 서비스로 분리해야 할까요?

대부분의 아키텍트는 이미 이러한 질문에 대한 답을 알고 있습니다. 바로 "상황에 따라 다르다"는 것입니다! 소프트웨어 아키텍처의 첫 번째 법칙으로 돌아가 보면, 소프트웨어 아키텍처의 모든 것은 절충의 문제입니다. 이러한 결정은 적용되는 특정 맥락을 포함하여 여러 요인에 따라 달라집니다. 모든 상황과 환경은 다르기 때문에 이러한 구조적 질문에 대한 "최선의 방법"이라는 것은 존재하지 않습니다.

대부분의 LLM(Learning Leadership Model)은 결과를 주로 확률에 기반하여 도출합니다. 즉, 주어진 맥락에서 가장 가능성이 높은 답은 무엇이며, 이 문제에 대한 "최적의 방법"은 무엇인가를 묻는 것입니다. 그러나 아키텍처 설계를 결정할 때는 확률이나 "최적의 방법"만으로는 충분하지 않습니다. 아키텍처 관련 질문에 답하려면 관련된 장단점을 신중하게 분석하고, 특정 비즈니스 및 기술적 맥락을 적용하여 가장 적절한 선택을 찾아야 합니다. 예를 들어, 비즈니스에서 가장 중요하게 생각하는 것이 시장 출시 시간(변경 사항 및 새로운 기능을 최대한 빨리 고객에게 제공하는 것)이라면, 성능보다 유지보수성이 훨씬 더 중요해지며, 의사결정 과정의 대부분은 유지보수성 최적화 방향으로 흘러갈 것입니다.

아키텍처 설계는 비즈니스 요구사항(예: 시장 출시 시간 단축 또는 지속 가능한 성장)을 아키텍처 특성(예: 유지보수성, 테스트 용이성, 배포 용이성 등)으로 변환하는 과정을 필요로 합니다. 이러한 변환은 항상 명확한 것은 아니며, 제대로 수행하려면 오랜 경험이 필요합니다. 일단 변환이 완료되면, 이는 트레이드오프 분석의 기초가 됩니다. 예를 들어, 결제 처리를 위한 단일 서비스를 사용할지, 아니면 결제 유형별로 서비스를 사용할지는 유지보수성과 성능 간의 절충점을 고려해야 합니다. 단일 서비스는 더 나은 성능을 제공하지만, 여러 서비스는 유지보수성을 향상시킵니다. 만약 비즈니스 요구사항이 주로 시장 출시 시간에 있다면, 유지보수성이 성능보다 훨씬 중요하므로, 이러한 특정 상황에서는 개별 서비스 방식이 적절한 선택이 될 것입니다.

개별적인 상황과 비즈니스 맥락을 분석하는 과정이 매우 특수하고 개별적이기 때문에, 현재의 생성형 AI로는 가장 적절한 아키텍처 설계를 도출하기 어렵습니다. 저자들이 최근 수행한 실험에 따르면, 최상의 시나리오는 생성형 AI 도구가 설계상의 가능한 절충안을 개략적으로 제시하여 간과된 절충안을 식별하는 데 도움을 주는 것입니다. 생성형 AI 도구는 풍부한 지식을 보유하고 있지만, 가장 적절한 아키텍처 설계를 내리는 데 필요한 지혜는 부족합니다.