CNF

2026.02

단종된 웹로직 서버의 기술적 위험과 오픈소스 WAS 전환 가이드

벤더 지원이 종료(EOS)된 **Oracle WebLogic Server** 운영 환경이 기업 IT 인프라에 미치는 심각한 기술 부채와 보안 위협을 심층 분석합니다.
단순히 버전 업그레이드를 넘어, 최신 Java 표준인 **Jakarta EE**와 **클라우드 네이티브 환경**으로 전환하기 위한 **오픈소스 WAS 마이그레이션 전략**과 실무 가이드를 제시합니다.

hello@cncf.co.kr

02-469-5426

www.cncf.co.kr

Contents

제1장. 단종된 WebLogic Server가 만드는 기술 부채의 실체

1.1 WebLogic Server 버전별 EOS(End of Support) 현황

단종된 Oracle WebLogic Server의 운영 환경은 IT 조직에 심각한 기술 부채를 야기합니다. 이 부채는 단순히 벤더의 기술 지원 종료를 넘어, 보안 패치 미제공, JDK 및 OS 지원 종료, 최신 Java EE/Jakarta EE 표준 적용 불가 등 다양한 연쇄적 제약으로 확대됩니다. 특히 WebLogic은 엔터프라이즈 Java 애플리케이션의 핵심 인프라로 오랜 기간 사용되어 왔으나, Oracle의 공식 지원 정책에 따라 각 버전별로 지원 종료(EOS) 일정이 엄격하게 적용되고 있습니다. Premier Support, Extended Support, Sustaining Support로 이어지는 3단계 지원 체계는 실무적으로 “지원은 되지만 고칠 수는 없는” 상태를 만들어, 보안·성능·호환성 측면에서 심각한 위험을 초래합니다. 본 절에서는 Oracle의 지원 정책과 WebLogic 버전별 EOS 현황, JDK와의 종속 구조를 상세히 분석하여, 단종된 WebLogic 환경이 왜 기술 부채의 근본 원인이 되는지 그 실체를 밝히고자 합니다.

1.1.1 Oracle Lifetime Support 정책과 지원 단계별 의미

Oracle의 제품 지원 정책은 Premier Support, Extended Support, Sustaining Support의 3단계로 구성되어 있습니다. Premier Support는 제품 출시 후 일정 기간 동안 제공되는 최상의 지원 단계로, 보안 패치, 버그 수정, 신규 인증 매트릭스(예: OS/JDK 호환성), 기능 개선 등이 모두 포함됩니다. Extended Support는 Premier Support가 종료된 후 추가 비용을 지불하면 받을 수 있는 연장 지원으로, 보안 패치와 주요 버그 수정은 제공되지만, 신규 기능이나 인증 매트릭스 업데이트는 제한됩니다.

가장 문제가 되는 단계는 Sustaining Support입니다. 이 단계에서는 기술 지원(문서, 기존 패치 다운로드 등)은 계속 제공되지만, 신규 보안 패치, 버그 수정, 인증 매트릭스 업데이트, 기능 개선이 모두 중단됩니다. 즉, CVE 취약점이 발견되어도 공식 패치가 제공되지 않으며, 새로운 OS/JDK와의 호환성 검증도 이루어지지 않습니다. 실무적으로 “지원은 되지만 고칠 수는 없는” 상태란, 시스템 장애나 보안 이슈가 발생해도 Oracle이 더 이상 수정해주지 않는다는 의미입니다.

예를 들어, WebLogic 10.3.6이나 12.1.3 등 Sustaining Only 상태에 들어간 버전에서는 최신 JDK나 OS로의 업그레이드가 불가능하며, 보안 취약점이 발견되어도 자체적으로 해결할 방법이 없습니다. 이러한 환경은 감사 지적, PCI DSS 등 컴플라이언스 위반, 외부 API 연동 거부 등 실질적인 운영 리스크로 이어집니다.

이처럼 Oracle의 지원 정책은 단종된 WebLogic Server가 기술 부채의 근본 원인이 되는 구조적 배경을 제공합니다. Sustaining Support 상태에서는 조직이 더 이상 시스템을 안전하게 유지할 수 없으며, 신규 개발이나 유지보수에도 심각한 제약이 발생합니다. 따라서 EOS 일정과 지원 단계별 의미를 정확히 이해하는 것이, 향후 마이그레이션 또는 대안 WAS 전환 전략 수립의 출발점이 됩니다.

1.1.2 WebLogic Server 7.0 이후 전 버전 EOS 일정표

Oracle 공식 Lifetime Support Policy 문서(MOS Note 950131.1)에 따르면, WebLogic Server의 각 버전은 아래와 같이 지원 종료(EOS) 일정을 갖고 있습니다. 이 일정은 JDK 지원 범위와도 밀접하게 연관되어, 기술 부채의 누적을 가속화합니다.

버전	GA 시점	Premier Support 종료	Extended Support 종료	현재 상태	지원 JDK
7.0 (BEA)	2002	2005.12	–	Sustaining Only	JDK 1.3/1.4
8.1 (BEA)	2003	2008.12	–	Sustaining Only	JDK 1.4
9.x	2006	2011.12	–	Sustaining Only	JDK 5
10.3.6 (11gR1)	2008	2014.01	2017.01	Sustaining Only	JDK 6, 7
12.1.3	2014	2019.12	2022.01	Sustaining Only	JDK 7, 8
12.2.1.3	2017	–	2023.06 (연장)	Sustaining Only	JDK 8
12.2.1.4 (LTS)	2019	2021.12	~2024 (Extended)	Extended/Sustaining	JDK 8
14.1.1	2020	~2025.03	TBD	Premier Support	JDK 8, 11
14.1.2 (예정)	2024~	TBD	TBD	미출시/GA 예정	JDK 17, 21

이 표에서 알 수 있듯이, WebLogic 7.0부터 12.2.1.3까지의 모든 버전은 Sustaining Only 상태에 들어가 있으며, 신규 보안 패치나 인증 매트릭스 업데이트가 중단된 상태입니다. 특히 WebLogic 10.3.6(11gR1)은 2017년 Extended Support 종료 후 Sustaining Only로 전환되

어, JDK 6/7만 지원 가능하며, JDK 8 이상으로 올릴 수 없습니다. 12.2.1.4(LTS) 역시 Extended Support가 2024년경 종료될 예정이며, 이후에는 Sustaining Only로 전환됩니다. 최신 버전인 14.1.1은 2025년 3월까지 Premier Support가 유지되지만, 이후 Extended/Sustaining 단계로 넘어갈 예정입니다.

이러한 EOS 일정은 실무적으로 “보안 패치 불가”, “JDK/OS 업그레이드 불가”, “최신 Java EE/Jakarta EE 적용 불가” 등 다양한 기술적 제약을 초래합니다. 특히, 지원 JDK 버전이 제한됨에 따라, 최신 Java 기능이나 성능 개선, 보안 강화 기능을 적용할 수 없는 구조적 한계가 발생합니다. 예를 들어, WebLogic 12.1.3은 JDK 7/8만 지원하므로 JDK 9 이상의 기능을 활용할 수 없고, 12.2.1.4 역시 JDK 8까지만 지원해 JDK 11/17의 새로운 GC, TLS, 언어 기능을 사용할 수 없습니다. 이러한 제한은 단순히 보안 패치의 부재에 그치지 않고, 전체 IT 인프라의 현대화와 혁신을 가로막는 장애물로 작용합니다. 실제로 많은 조직이 OS, JDK, WAS의 동시 업그레이드가 필요해지면서, 단일 계층의 업그레이드만으로는 기술 부채를 해소할 수 없는 상황에 직면하고 있습니다. 이로 인해 마이그레이션 프로젝트의 범위와 비용이 기하급수적으로 증가하며, 조직의 IT 전략 수립에 큰 부담을 주고 있습니다. 출처:Oracle Fusion Middleware Lifetime Support Policy, MOS Note 950131.1

1.1.3 WebLogic 버전과 JDK 버전 종속 관계

WebLogic Server의 각 버전은 지원 가능한 JDK 버전이 엄격하게 제한되어 있습니다. 이 구조는 WebLogic 버전이 JDK 버전의 “천장(ceiling)” 역할을 하여, 최신 JDK의 성능·보안 개선을 적용할 수 없게 만듭니다. 예를 들어, WebLogic 10.3.6은 JDK 6/7만 공식 지원하므로 JDK 8 이상을 사용할 수 없습니다. WebLogic 12.2.1.4는 JDK 8까지만 지원하며, JDK 11/17의 GC 개선(G1, ZGC 등), TLS 1.3 지원, 언어 기능 강화(Var, Stream API 등), 보안 강화 기능을 적용할 수 없습니다.

아래 표는 WebLogic 버전과 JDK 버전의 종속 구조를 요약한 것입니다.

WebLogic 버전	지원 JDK 버전	JDK 최신 기능 적용 가능 여부
10.3.6	6, 7	불가 (JDK 8 이상 미지원)
12.1.3	7, 8	JDK 9 이상 불가
12.2.1.3/4	8	JDK 11/17 불가

14.1.1	8, 11	JDK 17 미지원
14.1.2 (예정)	17, 21	최신 JDK 적용 가능 (GA 예정)

이 구조는 실무적으로 다음과 같은 문제를 야기합니다. 첫째, 보안 취약점 대응이 불가능해집니다. JDK 8 이전 버전은 TLS 1.2/1.3 등 최신 암호화 표준을 지원하지 않으므로, PCI DSS 등 컴플라이언스 위반이 발생합니다. 둘째, 최신 Java 언어 기능과 성능 개선(GC, 모듈 시스템 등)을 활용할 수 없어, 개발 생산성과 시스템 안정성이 저하됩니다. 셋째, WebLogic 버전 업그레이드 없이 JDK만 독립적으로 올리는 것이 불가능하므로, 전체 WAS+JDK+OS 계층을 동시에 교체해야 하는 대규모 마이그레이션이 필요합니다.

실제 현장에서는 WebLogic 10.3.6 환경에서 JDK 8로의 업그레이드를 시도하다가 공식 지원 부재로 인해 각종 호환성 오류와 예기치 못한 장애가 발생하는 사례가 빈번하게 보고되고 있습니다. 또한, JDK 11 이상의 GC 개선, TLS 1.3 지원, 모듈 시스템 도입 등 Java 생태계의 혁신적 변화가 WebLogic 구버전 환경에서는 전혀 적용되지 못하고 있습니다. 이로 인해 개발자들은 최신 언어 기능을 활용하지 못하고, 운영팀은 보안 취약점과 성능 저하에 지속적으로 노출됩니다. 결론적으로, WebLogic 버전이 JDK 버전의 천장이 되는 구조는 기술 부채의 누적을 가속화하며, 단종된 WebLogic 환경에서는 최신 Java 생태계와의 연동이 사실상 불가능합니다. 이러한 구조적 한계는 마이그레이션 또는 오픈소스 WAS 전환의 필요성을 더욱 강화합니다.

1.2 단종된 WebLogic 운영이 초래하는 기술적 문제 종합

단종된 WebLogic Server를 계속 운영하는 환경에서는 다양한 기술적 문제가 복합적으로 발생합니다. 이 문제들은 단순히 벤더 지원 종료에 그치지 않고, 보안, 컴플라이언스, 개발 생산성, 인력 수급, 라이선스 비용 등 IT 조직 전반에 심각한 영향을 미칩니다. 특히 Sustaining Only 상태에서는 보안 패치가 불가능하고, JDK 및 OS 지원 종료가 연쇄적으로 발생하며, Java EE 표준 정체로 인해 최신 프레임워크와의 호환성도 상실됩니다. 이로 인해 외부 결제 시스템, 금융 API, SaaS 연동 등 실질적인 서비스 장애가 발생할 수 있으며, 개발·운영 프로세스의 현대화도 저해됩니다. 본 절에서는 각 문제 영역별 구체적 증상과 실무 영향, 해결 난이도를 표와 사례 중심으로 분석합니다.

1.2.1 문제 영역별 상세 분석표

아래 표는 단종된 WebLogic 운영 환경에서 발생하는 주요 문제 영역을 구체적으로 정리한 것입니다.

문제 영역	구체적 증상	실무 영향	해결 난이도
보안 패치 불가	Sustaining 상태에서 CVE가 발견되어도 공식 패치 미제공	보안 취약점 노출 상태 고정, 감사 지적 반복	자체 해결 불가
TLS/암호화 표준 미달	TLS 1.0/1.1만 지원하는 구버전 JDK 사용	PCI DSS 미준수, 외부 API 연동 거부, 브라우저 접속 차단	WebLogic+JDK 동시 업그레이드 필요
JDK 버전 고정	WebLogic 10.3.x → JDK 6/7, 12.1.x → JDK 7/8 고정	최신 Java 언어 기능, 성능 개선(G1 GC, ZGC 등), 보안 강화 적용 불가	WebLogic 버전 업그레이드 없이 불가
Java EE 표준 정체	Java EE 5/6/7 수준에 고정, Jakarta EE 전환 불가	javax.* 네임스페이스 고정, 최신 프레임워크 호환성 상실	전면 마이그레이션 필요
OS 지원 종료 연쇄	CentOS 7 EOS(2024.06), RHEL 7 ELS 종료 등	구버전 WebLogic이 인증하는 OS가 단종되어 실행 환경 자체가 소멸	OS + WAS + JDK 동시 교체 필요
개발 생산성 저하	FTP 기반 배포, 형상관리 미연동, XML 중심 설정	CI/CD 파이프라인 구축 불가, 배포 오류·롤백 비용 증가	개발 프로세스 전면 개선 필요
기술 인력 수급 난	EJB 2.x, JSP/Servlet 2.x 기반 개발 경험자 감소	신규 채용 실패, 유지보수 인력 은퇴 시 기술 단절	기술 스택 전환 필요
라이선스 비용	사용하지 않는 기능 포함 연간 라이선스 유지	비용 대비 가치 하락, Sustaining 비용은 동일하게 청구	대안 WAS 전환으로 해결

구체적 사례와 실무 영향

- **보안 패치 불가**: 2023년 CVE-2023-21839와 같이 WebLogic의 T3/IIOP JNDI Lookup 취약점이 발견되어도, Sustaining Only 상태에서는 공식 패치가 제공되지 않습니다. 이로 인해 보안 취약점이 고정된 채 운영되며, 외부 감사에서 반복적으로 지적받게 됩니다.
- **TLS/암호화 표준 미달**: JDK 6/7만 지원하는 WebLogic 10.3.x 환경에서는 TLS 1.2/1.3을 사용할 수 없어, PCI DSS(2018.06 시한) 미준수로 금융 API 연동이 거부되고, 주요 브라우저(Chrome, Firefox 등)에서 접속이 차단됩니다.
- **JDK 버전 고정**: 최신 Java 언어 기능(예: Lambda, Stream API), GC 개선(G1, ZGC), 보안 강화 기능을 활용할 수 없어, 개발 생산성과 시스템 안정성이 저하됩니다.

- **Java EE 표준 정체**: javax.* 네임스페이스에 고정되어, Spring Boot 3.x, Hibernate 6.x 등 최신 프레임워크와의 호환성이 상실됩니다.
- **OS 지원 종료 연쇄**: CentOS 7, RHEL 7 등 WebLogic이 인증하는 OS가 2024년 EOS로 단종되면, 해당 OS용 JDK 보안 패치도 중단되어 WAS-JDK-OS 계층이 동시에 취약해집니다.
- **개발 생산성 저하**: FTP 기반 수동 배포, XML 중심 설정 등으로 CI/CD 파이프라인 구축이 불가능하며, 배포 오류 및 롤백 비용이 증가합니다.
- **기술 인력 수급 난**: EJB 2.x, JSP/Servlet 2.x 기반 개발 경험자가 급감하여 신규 채용이 실패하고, 기존 유지보수 인력 은퇴 시 기술 단절이 발생합니다.
- **라이선스 비용**: 사용하지 않는 기능까지 포함된 연간 라이선스 비용이 지속적으로 청구되며, Sustaining Only 상태에서도 비용은 동일하게 발생합니다.

이처럼 단종된 WebLogic 운영은 기술 부채의 누적을 가속화하며, 실무적으로 해결이 불가능한 구조적 문제를 야기합니다. 대안 WAS 전환이나 전면 마이그레이션이 불가피한 상황이 점차 확대되고 있습니다.

1.3 보안 관점의 위험 분석

단종된 WebLogic Server 환경에서 가장 심각한 위험은 보안 취약점의 지속적 노출입니다. WebLogic은 엔터프라이즈 환경에서 다양한 프로토콜(T3/IIOP, JNDI 등)을 지원하며, 복잡한 아키텍처로 인해 취약점이 반복적으로 발견되고 있습니다. 특히 Sustaining Only 상태에서는 신규 패치가 제공되지 않아, CVE 취약점이 고정된 채 운영될 수밖에 없습니다. 또한 WebLogic이 번들하거나 의존하는 Java 라이브러리(Log4j, Struts, Jackson, Spring 등)에서도 심각한 보안 이슈가 발생하며, 단종 버전에서는 라이브러리 교체가 기술적으로 어려운 구조적 한계가 존재합니다. 본 절에서는 WebLogic 주요 보안 취약점 연대기와 Java 플랫폼·오픈소스 라이브러리 취약점, TLS/암호화 표준 미달에 따른 컴플라이언스 위반을 분석합니다.

1.3.1 WebLogic Server 주요 보안 취약점 연대기 (2020~2025)

아래 표는 2020~2025년 WebLogic Server에서 발견된 주요 보안 취약점(CVE)과 공격 벡터, CVSS 점수, 영향받는 버전을 요약한 것입니다.

연도	CVE ID	CVSS	공격 벡터	영향
2020	CVE-2020-2883	9.8	T3 역직렬화 (ReflectionExtractor)	비인증 원격 코드 실행
2020	CVE-2020-14882/14750	9.8	HTTP Console 접근	비인증 RCE, 패치 우회
2021	CVE-2021-2394	9.8	T3/IIOP 역직렬화	비인증 원격 코드 실행
2022	CVE-2022-22965 (Spring4Shell)	9.8	Spring Framework (Third Party)	WebLogic 내장 라이브러리 경유 서버 탈취
2023	CVE-2023-21839	7.5	T3/IIOP JNDI Lookup	비인증 정보 유출 → RCE 연계
2024	14건 CVE 발표	평균 7.9	다양	지속적 취약점 노출
2025	2건 (평균 8.7)	8.7	-	위험도 상승 추세

특히 CVE-2020-14882와 CVE-2020-14750은 HTTP Console 접근 취약점으로, 패치가 우회되어 새로운 CVE가 반복적으로 발생하는 악순환이 이어졌습니다. T3/IIOP 프로토콜의 역직렬화 취약점(CVE-2020-2883, CVE-2021-2394 등)은 비인증 원격 코드 실행(RCE)로 이어져, 공격자가 서버를 완전히 탈취할 수 있는 심각한 위협입니다. Sustaining Only 상태에서는 이러한 취약점에 대한 공식 패치가 제공되지 않으므로, 단종 버전에서는 이 악순환을 끊을 방법이 없습니다. 실제로 2024년에도 14건 이상의 신규 CVE가 발표되었으며, 2025년에는 위험도가 더욱 상승하는 추세입니다.

이처럼 WebLogic 단종 환경은 보안 취약점이 반복적으로 노출되는 구조적 위험을 내포하고 있으며, 실무적으로는 외부 감사, 컴플라이언스 위반, 서비스 장애 등 심각한 운영 리스크로 이어집니다. 출처: Oracle Critical Patch Update Advisory, CVE Details, Rapid7, Tenable

1.3.2 Java 플랫폼 및 오픈소스 라이브러리 보안 취약점

WebLogic Server는 자체적으로 다양한 Java 라이브러리와 오픈소스 컴포넌트를 번들하거나 의존합니다. 단종된 WebLogic 환경에서는 이러한 라이브러리의 보안 취약점에 대응하기가 매우

어렵습니다.

- **Log4j (Log4Shell, CVE-2021-44228, CVSS 10.0)**: WebLogic 구버전에는 Log4j 1.x/2.x가 내장되어 있으며, Log4Shell 취약점이 발견되었음에도 단종 버전에서는 라이브러리 교체가 기술적으로 어렵습니다. 실제로 Log4j 2.x의 치명적 취약점은 서버 탈취로 이어질 수 있으며, WebLogic의 클래스로더 구조 때문에 애플리케이션 레벨에서 라이브러리를 교체해도 완전한 대응이 불가능한 경우가 많습니다.
- **Apache Struts (CVE-2017-5638 등)**: Equifax 사고 사례에서 보듯이, 프레임워크 버전 고정은 치명적 보안 사고로 이어질 수 있습니다. WebLogic 단종 환경에서는 Struts, Spring 등 번들 라이브러리의 업그레이드가 불가능하여, 취약점이 전파되는 구조적 위험이 존재합니다.
- **Jackson-databind 역직렬화**: 트랜지티브 의존성(transitive dependency)을 통해 취약점이 전파되며, WebLogic 내장 라이브러리와 애플리케이션 라이브러리 간 버전 불일치로 인한 NoSuchMethodError, ClassNotFoundException 등 런타임 오류가 빈번하게 발생합니다.
- **Spring Framework (Spring4Shell, CVE-2022-22965)**: WebLogic이 번들하는 Third Party 컴포넌트(Spring 등)를 경유한 공격 경로가 존재하며, 단종 버전에서는 라이브러리 교체가 불가능해 서버 탈취 위험이 상존합니다.

이처럼 WebLogic 단종 환경은 Java 플랫폼 및 오픈소스 라이브러리 취약점에 대한 대응이 구조적으로 불가능하며, OWASP Top 10의 “알려진 취약점이 있는 컴포넌트 사용” 위험이 상시 존재합니다.

1.3.3 TLS/암호화 표준 미달과 컴플라이언스 위반

구버전 JDK(5/6/7)는 TLS 1.2를 기본 지원하지 않으며, TLS 1.0/1.1만 지원하는 환경이 많습니다. PCI DSS에서는 2018년 6월 이후 TLS 1.0/1.1 사용을 금지하고 있으며, NIST SP 800-52에서도 TLS 1.2 이상 사용을 권고합니다. 주요 브라우저(Chrome, Firefox, Safari, Edge)는 2020년부터 TLS 1.0/1.1 연결을 거부하고 있습니다.

구체적으로, WebLogic 10.3.x와 JDK 6/7 조합에서는 TLS 1.2/1.3을 사용할 수 없어, 외부 결제 시스템, 금융 API, SaaS 연동이 물리적으로 불가능해집니다. 예를 들어, 금융권 API 연동 시

TLS 1.2 미지원으로 인해 인증이 거부되고, 브라우저에서 접속 자체가 차단됩니다. PCI DSS 준수 실패로 인해 외부 감사에서 반복적으로 지적받으며, 서비스 장애 및 고객 신뢰도 하락이 발생합니다.

이처럼 단종된 WebLogic+JDK 조합은 암호화 표준 미달로 인한 컴플라이언스 위반, 외부 연동 불가, 서비스 장애 등 실질적인 운영 리스크를 초래합니다. 이러한 구조적 한계는 WAS+JDK+OS 계층의 동시 교체 또는 오픈소스 WAS 전환이 불가피한 상황을 만듭니다.

출처

- Oracle Fusion Middleware Lifetime Support Policy (https://www.oracle.com/us/assets/lifetime-support-middleware-069163.pdf), MOS Note 950131.1
- Oracle Critical Patch Update Advisory, CVE Details (https://www.cvedetails.com/product/14534/Oracle-Weblogic-Server.html), Rapid7, Tenable
- PCI DSS, NIST SP 800-52

제2장. WebLogic 종속 기술의 단종과 연쇄적 기술 제약

2.1 운영체제 EOS와 JDK 지원 범위의 연쇄 효과

WebLogic 환경에서 운영체제와 JDK의 지원 종료는 단일 계층의 문제가 아니라, 전체 시스템에 연쇄적인 영향을 미치는 중요한 이슈입니다. 특히, WebLogic은 특정 운영체제와 JDK 버전에서만 공식적으로 인증되어 동작하므로, 이들 중 하나라도 지원이 종료될 경우 전체 시스템의 보안성, 안정성, 그리고 규제 준수 능력이 심각하게 저하됩니다. 이러한 연쇄 효과는 단순히 기술적 불편을 넘어, 실제 서비스 중단, 외부 감사 지적, 비용 증가 등 실질적인 경영 리스크로 이어질 수 있습니다. 본 절에서는 운영체제와 JDK의 EOS(End of Support) 현황, 그리고 이로 인해 발생하는 WebLogic 환경의 제약과 실무적 위험을 구체적으로 살펴봅니다.

2.1.1 단종된 OS에서의 JDK 지원 제한

WebLogic Server의 인증 OS 목록은 버전별로 이미 EOS에 도달한 경우가 많습니다. WebLogic 7, 8, 9 버전은 물론, 10.3.6(11g), 12.1.x, 12.2.1.x 등도 RHEL 5/6, CentOS 6/7, Solaris 10/11 등 단종된 운영체제에서만 공식적으로 지원됩니다. 예를 들어, WebLogic 10.3.6은 RHEL 6과 CentOS 6/7, Solaris 10에서만 인증되었으나, RHEL 6은 2020년 11월, CentOS 7은 2024년 6월, Solaris 10/11은 Sustaining 상태로 사실상 지원이 종료되었습니다. 이와 같이 OS가 단종되면 해당 OS용 JDK 역시 보안 패치와 기능 개선이 중단되어, WAS-JDK-OS 세 계층이 모두 취약해지는 구조가 됩니다.

OS 버전	EOS 일자	영향받는 WebLogic 버전	지원 JDK 범위
RHEL 6	2020.11	10.3.6, 12.1.x	JDK 6, 7
CentOS 7	2024.06	10.3.6, 12.1.x, 12.2.1.x	JDK 7, 8
RHEL 7	2024.06 (ELS ~2028)	12.1.x, 12.2.1.x	JDK 8
Solaris 11	Sustaining	10.3.6 ~ 14.1.1	JDK 8, 11

실무적으로는 OS 단종에 따라 보안 패치 적용이 불가능해지고, JDK 역시 해당 OS에서만 동작하는 버전으로 고정됩니다. 예를 들어, RHEL 6에서 JDK 8은 더 이상 업데이트되지 않으며, RHEL 7 역시 2024년 6월 이후 ELS(유료 연장)만 제공됩니다. 이로 인해 WebLogic 환경은 최신 JDK 기능(GC 개선, TLS 1.2/1.3 등)과 보안 패치(CVE 대응)를 적용할 수 없게 됩니다. 또한, OS 단종은 인프라 운영팀의 기술 지원, 클라우드 전환, 외부 연동에도 심각한 제약을 가져오며, 실질적으로 시스템 전체의 생명주기가 종료되는 결과를 초래합니다.

구체적인 사례로, 금융기관의 WebLogic 10.3.6 환경이 CentOS 7에서 운영되고 있다면, 2024년 6월 이후 OS 보안 패치가 제공되지 않아 PCI DSS 등 규제 준수에 실패하게 됩니다. JDK 역시 7/8로 고정되어 최신 TLS 표준을 지원하지 못하며, 외부 API 연동이나 브라우저 접속이 차단될 수 있습니다. 이처럼 WAS-JDK-OS 계층의 연쇄 단종은 단순한 기술적 불편을 넘어 실무적 위험과 비용 증가로 이어집니다.

이외에도, OS 단종으로 인한 기술 지원 중단은 장애 발생 시 신속한 대응이 어려워지는 문제를 야기합니다. 예를 들어, OS 커널 취약점이나 시스템 라이브러리의 보안 이슈가 발견되더라도 공식 패치가 제공되지 않으므로, 운영팀은 자체적으로 임시 방편을 마련하거나, 비공식 패치를 적용해야

하는 상황에 직면하게 됩니다. 이 과정에서 시스템의 안정성이 저하되고, 예기치 않은 장애가 발생할 가능성도 높아집니다. 또한, 클라우드 환경으로의 전환을 고려할 때, 단종된 OS는 주요 클라우드 서비스 제공업체에서 지원하지 않으므로, 마이그레이션 자체가 불가능하거나 추가적인 커스텀 작업이 필요하게 됩니다. 이러한 복합적인 제약은 결국 조직의 IT 전략 수립과 실행에 큰 부담으로 작용합니다.

2.1.2 JDK 버전별 EOS 현황과 WebLogic 영향

JDK의 지원 종료(EOS) 일정은 WebLogic 환경에 직접적인 영향을 미칩니다. Oracle JDK 6은 Premier Support가 2015년 12월에 종료되었고, JDK 7은 2019년 7월, JDK 8은 Extended Support(유료)로 2030년 12월까지 제공되지만, 무료 커뮤니티 지원(OpenJDK 8/11)은 2025년 7월에 종료됩니다. 특히, WebLogic 10.3.x는 JDK 6/7, 12.1.x는 JDK 7/8까지만 지원하므로, 최신 JDK(11, 17, 21 등)로의 업그레이드가 구조적으로 불가능합니다.

JDK 버전	Oracle JDK Premier 종료	OpenJDK 커뮤니티 종료	WebLogic 지원 범위
JDK 6	2015.12	2016.12	10.3.x
JDK 7	2019.07	2020.06	10.3.x, 12.1.x
JDK 8	2022.03 (무료)	2025.07	12.1.x, 12.2.1.x
JDK 8 (유료)	2030.12	2030.12 (Temurin)	12.2.1.x
JDK 11	2024.10	2024.10	14.1.x
JDK 17	2027.10	2027.10	14.1.2 (예정)
JDK 21	2029.12	2029.12	14.1.2 (예정)

이처럼 구버전 WebLogic 사용자는 무료 보안 패치를 받을 수 있는 JDK가 2025년 7월 이후에는 없어지게 됩니다. 실무적으로는 JDK 8 Extended(유료)로만 보안 패치가 제공되며, WebLogic 12.2.1.4 이하 버전에서는 JDK 11/17/21의 성능·보안 개선을 적용할 수 없습니다. 또한, OpenJDK 커뮤니티 지원 종료로 인해, 보안 취약점 대응이 불가능해지고, 외부 감사나 규제 준수에 실패할 위험이 높아집니다.

실제 사례로, 공공기관의 WebLogic 12.1.3 환경이 JDK 8에서 운영되고 있다면, 2025년 7월 이후 무료 보안 패치가 중단되어 CVE 대응이 불가능해집니다. Oracle JDK Extended

Support(유료)를 구매하지 않는 한, 보안 취약점에 노출된 상태가 고정되며, 외부 연동이나 클라우드 전환에도 심각한 제약이 발생합니다. 이처럼 JDK EOS 일정은 WebLogic 환경의 생명주기와 직결되며, 기술 부채와 보안 위험을 가속화합니다.

JDK EOS의 또 다른 문제점은, JDK 버전별로 제공되는 기능과 성능 최적화의 차이입니다. 예를 들어, JDK 11 이후에는 GC(가비지 컬렉션) 알고리즘의 개선, TLS 1.3 지원, Flight Recorder 등 다양한 성능 및 보안 기능이 추가되었으나, 구버전 WebLogic은 이러한 최신 기능을 활용할 수 없습니다. 또한, JDK 8 이후에는 모듈 시스템(JPMS), 새로운 HTTP 클라이언트 API, 향상된 JVM 모니터링 도구 등이 도입되었으나, WebLogic 12.2.1.4 이하 환경에서는 이러한 기능을 사용할 수 없어, 운영 효율성과 개발 생산성이 저하됩니다. 더불어, JDK의 보안 취약점(CVE)이 지속적으로 발견되고 있으나, EOS 이후에는 공식 패치가 제공되지 않으므로, 시스템이 영구적으로 보안 위협에 노출되는 구조적 한계가 발생합니다. 이러한 이유로, JDK EOS는 단순한 소프트웨어 지원 종료가 아니라, 전체 WebLogic 기반 시스템의 생명주기와 미래 전략에 결정적인 영향을 미치는 요소임을 명확히 인식해야 합니다.

2.2 Java EE에서 Jakarta EE로의 전환과 기술 표준 단절

Java EE에서 Jakarta EE로의 전환은 최근 20년간 Java 생태계에서 가장 큰 호환성 변화로 평가받고 있습니다. Oracle이 Java EE를 Eclipse Foundation에 이관하면서, javax.* 네임스페이스의 진화 권한이 사라지고, Jakarta EE 9(2020.12)부터 모든 EE API가 jakarta.*로 변경되었습니다. 이로 인해 기존 WebLogic 환경에서는 최신 프레임워크(Spring Framework 6.x, Spring Boot 3.x, Hibernate 6.x, MicroProfile 6.x 등)와의 호환성이 구조적으로 단절됩니다. WebLogic 12.2.1.4 이하는 Java EE 7까지만 지원하며, WebLogic 14.1.1도 Jakarta EE 8(여전히 javax.* 네임스페이스)까지만 지원하므로, jakarta.* 네임스페이스를 사용하는 최신 기술을 적용할 수 없습니다. 이로 인해 코드 레벨에서 import 문, 설정 파일, ServiceLoader 등 다양한 부분에서 호환성 문제가 발생하며, 실무적으로는 전면 마이그레이션이 필요해집니다.

Java EE와 Jakarta EE의 전환은 단순히 네임스페이스 변경에 그치지 않고, Java 기반 엔터프라이즈 애플리케이션의 개발, 배포, 유지보수 방식 전반에 걸쳐 광범위한 영향을 미치고 있습니다. 특히, 기존 WebLogic 환경에서는 최신 오픈소스 프레임워크와의 호환성 단절, 라이브러리

의존성 문제, 그리고 코드 및 설정 파일의 대규모 수정이 불가피해졌습니다. 이러한 변화는 조직의 기술 전략, 인력 운영, 그리고 장기적인 시스템 아키텍처에 중대한 영향을 미치므로, 본 절에서는 네임스페이스 변경의 기술적 의미, WebLogic 환경에서의 Jakarta EE 적용 불가 구조, 그리고 레거시 라이브러리 의존성 문제를 심층적으로 분석합니다.

2.2.1 javax.* → jakarta.* 네임스페이스 변경의 기술적 의미

Oracle이 Java EE를 Eclipse Foundation에 이관하면서, Java EE의 API 진화 권한이 Eclipse로 넘어갔습니다. 이 과정에서 라이선스 정책상 javax.* 네임스페이스를 더 이상 변경할 수 없게 되었고, Jakarta EE 9부터 모든 EE API가 jakarta.*로 변경되었습니다. 이 변화는 "지난 20년간 Java 생태계에서 가장 큰 호환성 변경"으로 평가받고 있습니다. 기존 Java EE 5/6/7 기반의 애플리케이션은 javax.* 네임스페이스를 사용하지만, Jakarta EE 9/10 이후에는 jakarta.* 네임스페이스가 표준이 되었습니다.

기술적으로는 import 문이 아래와 같이 변경됩니다.

```
// Java EE 7
import javax.servlet.*;
import javax.persistence.*;
// Jakarta EE 10
import jakarta.servlet.*;
import jakarta.persistence.*;
```

이 변화는 단순한 네임스페이스 변경을 넘어, 라이브러리 호환성, 설정 파일, ServiceLoader, Annotation 처리 등 다양한 부분에서 호환성 문제를 야기합니다. 예를 들어, Spring Framework 6.x, Hibernate 6.x, MicroProfile 6.x 등 최신 프레임워크는 jakarta.* 네임스페이스를 요구하므로, 구버전 WebLogic 환경에서는 적용이 불가능합니다. 실무적으로는 기존 애플리케이션의 import 문, 설정 파일, 의존성 트리 전체를 전면 수정해야 하며, 이는 대규모 코드 리팩토링과 테스트를 필요로 합니다.

네임스페이스 변경의 파급 효과는 단순히 코드 수정에만 국한되지 않습니다. 예를 들어, 기존에 javax.servlet API를 기반으로 작성된 필터, 리스너, 서블릿 클래스들은 jakarta.servlet API로의 전환이 필요하며, 이 과정에서 컴파일 오류 및 런타임 오류가 빈번하게 발생할 수 있습니다. 또한, Maven, Gradle 등 빌드 도구의 의존성 관리 설정에서도 javax.* 기반 라이브러리를 jakarta.* 기반 라이브러리로 교체해야 하며, 이로 인해 의존성 충돌이나 트랜지티브 의존성 문제도 발생할

수 있습니다. 설정 파일(web.xml, persistence.xml 등)에서도 네임스페이스 변경이 요구되며, 일부 프레임워크의 경우 설정 파일 내에서 명시적으로 네임스페이스를 지정해야 정상 동작합니다. ServiceLoader 기반 확장 기능이나, 어노테이션 프로세서 등도 네임스페이스 변경에 따라 동작 방식이 달라질 수 있으므로, 전체 시스템 차원에서의 영향 분석과 테스트가 필수적입니다.

이처럼 javax.*에서 jakarta.*로의 네임스페이스 변경은 단순한 문법적 차원이 아니라, 엔터프라이즈 Java 생태계 전반에 걸친 구조적 변화임을 명확히 인식해야 하며, 조직 차원의 전략적 대응이 필요합니다.

2.2.2 단종된 WebLogic에서의 Jakarta EE 적용 불가 구조

WebLogic 12.2.1.4 이하는 Java EE 7까지만 지원하며, WebLogic 14.1.1도 Jakarta EE 8(여전히 javax.* 네임스페이스)까지만 지원합니다. Jakarta EE 9부터는 모든 API가 jakarta.* 네임스페이스로 변경되어, 최신 프레임워크(Spring Framework 6.x, Spring Boot 3.x, Hibernate 6.x, MicroProfile 6.x 등)는 구버전 WebLogic에서 사용할 수 없습니다. 이는 코드 레벨에서 import 문, 설정 파일, ServiceLoader, Annotation 등 다양한 부분에서 호환성 문제가 발생함을 의미합니다.

구체적으로, Spring Boot 3.x는 jakarta.* 네임스페이스를 요구하므로, WebLogic 12.2.1.4 환경에서는 아래와 같은 오류가 발생합니다.

```
// WebLogic 12.2.1.4에서는 ClassNotFoundException 발생
import jakarta.servlet.*;
```

설정 파일에서도 javax.* → jakarta.* 변경이 필요하며, ServiceLoader 기반의 확장 기능도 호환되지 않습니다. 실무적으로는 기존 애플리케이션의 전체 코드와 설정을 전면 수정해야 하며, 이는 대규모 리팩토링과 테스트를 필요로 합니다. 또한, WebLogic 전용 기술자(weblogic.xml, weblogic-ejb-jar.xml 등) 역시 최신 Jakarta EE 표준과 호환되지 않아, 전면 마이그레이션이 불가피합니다.

이처럼 단종된 WebLogic 환경에서는 Jakarta EE 전환이 구조적으로 불가능하며, 최신 프레임워크와의 호환성 단절이 발생합니다. 실무적으로는 신규 개발은 오픈소스 WAS(Spring Boot, WildFly, Open Liberty 등) 기반으로 진행하고, 기존 애플리케이션은 점진적으로 이전하는 전략이 필요합니다.

이러한 구조적 한계는 단순히 코드 호환성 문제에 그치지 않고, 조직의 장기적인 IT 전략에도 영향을 미칩니다. 예를 들어, 최신 보안 표준이나 클라우드 네이티브 아키텍처를 도입하려 해도, Jakarta EE 기반의 프레임워크 및 라이브러리를 사용할 수 없으므로, DevOps, CI/CD, 컨테이너화 등 현대적 개발·운영 방식의 도입이 제한됩니다. 또한, 외부 벤더나 오픈소스 커뮤니티의 지원을 받기 어렵고, 신규 개발자들이 익숙한 최신 기술 스택과의 괴리가 커져 인력 확보에도 어려움이 따릅니다. 이로 인해, 기존 WebLogic 환경을 유지하는 조직은 점차 기술 부채가 누적되고, 시스템의 확장성, 유연성, 보안성 측면에서 경쟁력을 상실할 위험이 높아집니다. 따라서, Jakarta EE 전환이 불가능한 WebLogic 환경은 장기적으로 조직의 IT 경쟁력에 심각한 제약을 초래할 수밖에 없습니다.

2.2.3 레거시 Java 라이브러리 의존성의 보안·호환성 문제

구버전 WebLogic 환경에서는 라이브러리 버전이 고정되어 있어, 트랜지티브 의존성(transitive dependency)을 통한 취약점 전파, 라이브러리 버전 불일치로 인한 NoSuchMethodError, ClassNotFoundException 등 다양한 런타임 오류가 발생합니다. 또한, WebLogic 내장 라이브러리와 애플리케이션 라이브러리 간의 클래스로더 충돌이 빈번하게 발생하며, OWASP Top 10에 포함된 “알려진 취약점이 있는 컴포넌트 사용” 위험에 노출됩니다.

실무적으로, WebLogic은 자체적으로 여러 라이브러리(예: Log4j, Jackson, Apache Commons 등)를 내장하고 있으며, 애플리케이션(WAR/EAR)은 WEB-INF/lib에 별도의 라이브러리를 포함합니다. 동일한 라이브러리가 서버와 WAR 양쪽에 존재하면, 버전 충돌이나 클래스 로딩 순서 문제로 인해 NoSuchMethodError, ClassNotFoundException, NoClassDefFoundError 등이 발생합니다. 특히, 트랜지티브 의존성으로 인해, 애플리케이션이 직접 참조하지 않는 라이브러리에서도 취약점이 전파될 수 있습니다.

예를 들어, Log4j 1.x/2.x의 취약점(Log4Shell, CVE-2021-44228 등)은 WebLogic 내장 라이브러리와 애플리케이션 라이브러리 간 버전 불일치로 인해, 패치 적용이 어려워지고, 보안 취약점이 해결되지 않은 상태로 고정됩니다. 또한, JGroups, Infinispan, SLF4J 등 클러스터링이나 로깅 라이브러리의 버전 불일치로 인해, 클러스터 통신 장애, 로그 미출력 또는 중복 출력 등 다양한 문제가 발생합니다.

클래스로더 충돌은 WebLogic의 독자적인 계층 구조(System → Domain → Application →

Module)와 애플리케이션의 라이브러리 구조가 다르기 때문에 발생합니다. 실무적으로는 prefer-web-inf-classes, prefer-application-packages 등 WebLogic 전용 클래스로더 정책을 사용하지만, 오픈소스 WAS(WildFly, Tomcat 등)에서는 jboss-deployment-structure.xml, context.xml 등 별도의 설정이 필요합니다.

이처럼 레거시 Java 라이브러리 의존성은 보안 취약점, 호환성 문제, 런타임 오류 등 다양한 위험을 초래하며, 실무적으로는 전면적인 라이브러리 버전 관리와 클래스로더 정책 변경이 필요합니다. OWASP Top 10에서 “알려진 취약점이 있는 컴포넌트 사용”이 주요 위험으로 지적되는 만큼, WebLogic 환경에서는 라이브러리 버전 관리와 보안 패치 적용이 사실상 불가능해지는 구조적 한계가 있습니다.

더불어, 레거시 환경에서는 라이브러리 업그레이드가 쉽지 않기 때문에, 취약점이 발견된 라이브러리를 최신 버전으로 교체하려 해도, WebLogic의 내장 라이브러리와의 호환성 문제로 인해 실제 적용이 어렵습니다. 예를 들어, Log4j 2.x로의 업그레이드가 필요하더라도, WebLogic이 내부적으로 Log4j 1.x를 사용하고 있다면, 새로운 버전의 Log4j를 추가하는 순간 클래스 충돌이나 예기치 않은 동작이 발생할 수 있습니다. 또한, 라이브러리의 트랜지티브 의존성으로 인해, 직접적으로 사용하지 않는 라이브러리에서 발생하는 취약점까지도 관리해야 하므로, 전체 애플리케이션의 보안 관리가 매우 복잡해집니다. 이러한 문제는 보안 감사나 외부 규제 대응 시에도 큰 장애 요인으로 작용하며, 실제로 여러 금융기관이나 공공기관에서 라이브러리 취약점으로 인한 감사 지적 사례가 다수 보고되고 있습니다. 따라서, 레거시 WebLogic 환경에서는 라이브러리 의존성 관리와 보안 패치 적용이 구조적으로 어려워, 장기적으로는 오픈소스 WAS 기반의 현대적 아키텍처로의 전환이 필수적입니다.

2.3 EJB 기반 아키텍처의 구조적 한계

EJB(Enterprise JavaBeans)는 Java EE 초기에 엔터프라이즈 컴포넌트 모델로 설계되어, 대규모 분산 트랜잭션, 상태 관리, 메시징 등 복잡한 비즈니스 로직 구현에 활용되었습니다. 그러나 Spring Framework의 등장 이후, EJB의 복잡성과 성능 한계, 개발 생산성 저하 등으로 인해 사용이 급감하였습니다. 2025년 기준 Java 개발자 설문(JRebel 등)에서 EJB 사용 비율은 극히 낮은 수준으로 나타나고 있으며, 신규 개발자들은 EJB 경험이 거의 없는 현실입니다. 이로 인해

EJB 기반 애플리케이션의 유지보수와 인력 수급이 심각한 문제로 대두되고 있습니다.

EJB 기반 아키텍처는 과거 대규모 엔터프라이즈 시스템에서 표준으로 자리 잡았으나, 최근에는 그 구조적 한계와 기술 부채로 인해 점차 도태되고 있습니다. 특히, 복잡한 설정과 배포 기술자, 낮은 개발 생산성, 그리고 인력 수급의 어려움은 조직의 IT 운영에 큰 부담을 주고 있습니다. 본 절에서는 EJB 2.x/3.x 기술의 현재 위치, EJB 의존 애플리케이션의 유지보수 어려움 등 실무적 관점에서 EJB 기반 아키텍처의 한계와 그로 인한 조직적·기술적 문제를 상세히 분석합니다.

2.3.1 EJB 2.x/3.x 기술의 현재 위치

EJB는 1999년 Java EE 초기에 엔터프라이즈 컴포넌트 모델로 설계되어, 대규모 트랜잭션, 상태 관리, 메시징 등 복잡한 비즈니스 로직 구현에 활용되었습니다. EJB 2.x는 Entity Bean, Session Bean, Message-Driven Bean 등 다양한 컴포넌트 모델을 제공했으나, 개발 복잡성과 성능 한계로 인해 실무에서는 사용이 급감하였습니다. Spring Framework의 등장 이후, EJB의 복잡한 설정과 XML 중심 개발 방식, 배포 기술자(weblogic-ejb-jar.xml 등)의 이식성 부재가 단점으로 지적되었고, Spring의 POJO 기반 개발, DI, AOP 등 현대적 개발 패러다임이 대세가 되었습니다.

2025년 기준 Java 개발자 설문(JRebel 등)에서 EJB 사용 비율은 5% 미만으로 나타나고 있으며, 신규 개발자들은 EJB 경험이 거의 없는 현실입니다. 실무적으로는 EJB 2.x Entity Bean이 JPA(Entity)로 대체되었고, Stateless/Stateful Session Bean은 Spring @Service, @SessionScope 등으로 전환되었습니다. Message-Driven Bean(MDB) 역시 JMS Listener 또는 Spring @JmsListener로 대체되고 있습니다.

이처럼 EJB는 현재 엔터프라이즈 환경에서 점진적으로 퇴출되고 있으며, 신규 개발은 대부분 Spring Boot, Jakarta EE 기반으로 진행됩니다. 기존 EJB 기반 애플리케이션은 유지보수와 인력 수급에 심각한 문제를 겪고 있으며, 전면적인 기술 스택 전환이 요구되는 상황입니다.

EJB의 현재 위치를 좀 더 구체적으로 살펴보면, EJB 3.x 이후에는 어노테이션 기반 개발이 도입되어 사용성이 다소 개선되었으나, 여전히 Spring Framework의 간결함과 생산성에는 미치지 못합니다. 또한, EJB는 WAS 종속성이 강해, WebLogic, JBoss, WebSphere 등 특정 WAS에 맞춘 배포 기술자와 설정 파일이 필요하므로, 이식성과 확장성이 떨어집니다. 최근에는 클라우드 네이티브 환경, 마이크로서비스 아키텍처가 대세가 되면서, EJB의 무거운 컨테이너 기반 구조가 오히려 장애 요인으로 작용하고 있습니다. 실제로, EJB 기반 시스템을 운영 중인 조직에서는 신규

기능 개발이나 장애 대응 시, EJB 구조의 복잡성으로 인해 개발 기간이 늘어나고, 테스트 및 배포 과정에서 예기치 않은 문제가 자주 발생하는 사례가 보고되고 있습니다. 이러한 이유로, EJB는 점차 레거시 기술로 인식되고 있으며, 조직 차원에서 Spring Boot, Jakarta EE 등 현대적 기술 스택으로의 전환이 가속화되고 있습니다.

2.3.2 EJB 의존 애플리케이션의 유지보수 어려움

EJB 2.x Entity Bean은 EJB 3.x에서 개념 자체가 제거되어, JPA(Entity)로 전환이 필요합니다. 그러나 실무적으로는 Entity Bean의 복잡한 설정, 배포 기술자(weblogic-ejb-jar.xml, weblogic-cmp-rdbms-jar.xml 등)의 이식성 부재, JNDI Lookup 하드코딩 등 다양한 문제로 인해 전환이 매우 어렵습니다. Stateful Session Bean은 상태 관리 방식이 근본적으로 달라, Spring @SessionScope, CDI @SessionScoped 등으로 전환 시 설계 변경이 필요합니다. Message-Driven Bean(MDB)은 WebLogic 전용 설정(weblogic-ejb-jar.xml 등)을 JMS Listener 또는 Spring @JmsListener로 변환해야 하며, 이는 대규모 코드 수정과 테스트를 필요로 합니다.

실무적으로는 EJB 경험 개발자 수급이 사실상 불가능하며, 채용 시장에서도 EJB 경험자는 극히 드물게 나타납니다. 유지보수 인력이 은퇴하거나 이직할 경우, 기술 단절이 발생하여 시스템 전체의 안정성이 저하됩니다. 또한, WebLogic 전용 배포 기술자와 JNDI Lookup 하드코딩은 오픈소스 WAS(WildFly, Open Liberty 등)로의 전환 시 이식성이 매우 낮아, 전면적인 코드 리팩토링이 필요합니다.

구체적인 사례로, 금융기관의 EJB 2.x 기반 애플리케이션은 Entity Bean → JPA Entity 전환이 불가능하며, Stateful Session Bean의 상태 관리 복잡성으로 인해 설계 변경이 불가피합니다. WebLogic 전용 기술자와 JNDI Lookup 하드코딩은 오픈소스 WAS로의 마이그레이션 시 장애 요인으로 작용하며, 실무적으로는 신규 개발은 Spring Boot 기반으로 진행하고, 기존 EJB 애플리케이션은 점진적으로 이전하는 전략이 필요합니다.

EJB 의존 애플리케이션의 유지보수 어려움은 단순히 기술적 문제에 그치지 않습니다. 예를 들어, EJB 2.x 기반 시스템에서는 비즈니스 로직이 여러 XML 설정 파일과 분산된 자바 클래스에 복잡하게 얽혀 있어, 신규 인력이 시스템 구조를 파악하는 데 상당한 시간이 소요됩니다. 또한, JNDI Lookup 하드코딩은 리팩토링 시 일괄 변경이 어렵고, WAS별로 JNDI 네이밍 규칙이 달라

마이그레이션 과정에서 예기치 않은 오류가 발생할 수 있습니다. 배포 기술자(weblogic-ejb-jar.xml, weblogic-cmp-rdbms-jar.xml 등)는 WebLogic에 특화된 속성들이 많아, 오픈소스 WAS로 이전할 때 대부분의 설정을 새롭게 작성해야 하며, 이 과정에서 데이터베이스 매핑, 트랜잭션 관리, 보안 설정 등도 모두 재설계가 필요합니다. 더불어, EJB 기반 시스템은 테스트 자동화가 어렵고, CI/CD 파이프라인 구축에도 제약이 많아, 현대적 개발·운영 방식 도입에 장애가 됩니다. 실제로, 여러 금융·공공기관에서 EJB 유지보수 인력 부족과 기술 단절로 인해, 시스템 장애 발생 시 신속한 대응이 어려워지는 사례가 빈번하게 보고되고 있습니다. 이러한 복합적인 어려움으로 인해, EJB 의존 애플리케이션은 장기적으로 유지보수 비용이 급증하고, 조직의 IT 경쟁력 저하로 이어질 수밖에 없습니다.

제3장. 마이그레이션 전략 비교: WebLogic 업그레이드 vs. 오픈소스 WAS 전환

3.1 WebLogic 상위 버전 업그레이드 방식

WebLogic Server는 오랜 기간 엔터프라이즈 환경에서 핵심 WAS(Web Application Server)로 활용되어 왔으나, 최근 단종 및 지원 종료(EOS) 이슈로 인해 상위 버전 업그레이드 또는 대체 기술 전환이 필수적인 상황에 직면하고 있습니다. 이 장에서는 WebLogic 상위 버전 업그레이드의 구조와 현실적 제약, 그리고 오픈소스 WAS로의 전환과 비교하여 마이그레이션 전략을 심도 있게 분석합니다. WebLogic 업그레이드는 기존 시스템의 연속성과 Oracle 벤더 지원을 유지할 수 있다는 장점이 있지만, JDK 및 Java EE/Jakarta EE 표준의 최신 기능 적용에 한계가 있으며, 라이선스 비용과 벤더 종속성이 지속된다는 단점이 존재합니다. 반면 오픈소스 WAS 전환은 비용 절감과 기술 유연성을 제공하지만, 기존 WebLogic 전용 기능의 대체 구현과 코드 리팩토링이 필요합니다. 본 절에서는 WebLogic 내 업그레이드 경로, 각 단계별 기술적 제약, 그리고 업그레이드 방식의 장단점을 구체적으로 설명합니다.

3.1.1 WebLogic 내 버전 업그레이드 경로와 제약

WebLogic Server의 업그레이드 경로는 일반적으로 다음과 같은 단계로 진행됩니다: WebLogic 7, 8, 9 → WebLogic 10.3.6 → 12.2.1.4 → 14.1.1 → 14.1.2(예정). 각 단계마다 JDK 및 Java EE/Jakarta EE 스펙 변경, 배포 기술자 구조 변화, API 비호환성 등 다양한 기술적 제약이 발생합니다.

먼저, WebLogic 7, 8, 9 버전은 BEA Systems 시절의 제품으로 JDK 1.3/1.4에 종속되어 있으며, Java EE 5 이전의 표준을 지원합니다. 이들 버전은 이미 Sustaining Support 상태로, 보안 패치와 버그 수정이 중단되었습니다. WebLogic 10.3.6(11gR1)은 JDK 6/7을 지원하며, Java EE 5를 기반으로 하고 있습니다. 이 버전에서 12.2.1.4로 업그레이드할 경우 JDK 8과 Java EE 7까지 지원 범위가 확장됩니다. 그러나 이 과정에서 배포 기술자(weblogic.xml, weblogic-ejb-jar.xml 등)와 API 사용 패턴이 변경되어, 코드 수정이 필수적입니다.

특히 WebLogic 10.3.6에서 12.x로의 전환은 Java EE 5에서 6/7로의 스펙 변경이 이루어지므로, EJB, JPA, Servlet, JSP 등 주요 컴포넌트에서 대규모 코드 리팩토링이 필요합니다. 예를 들어, EJB 2.x Entity Bean에서 JPA Entity로의 전환, JNDI Lookup 방식의 변경, 트랜잭션 관리 방식의 업데이트 등이 요구됩니다. 또한, JDK 버전 변경에 따라 기존 코드의 호환성 문제가 발생할 수 있으며, 신규 JDK의 보안 및 성능 개선(G1 GC, TLS 1.2 등)을 적용하려면 코드와 설정을 재검토해야 합니다.

WebLogic 12.2.1.4에서 14.1.1로 업그레이드할 경우 Jakarta EE 8까지 지원하지만, 네임스페이스가 javax.*에 고정되어 있어 최신 Jakarta EE 9/10의 기능(jakarta.* 네임스페이스) 적용이 불가능합니다. 14.1.2(예정) 버전부터 JDK 17/21과 Jakarta EE 10 지원이 가능하나, 실제 GA 일정과 Oracle의 인증 매트릭스에 따라 업그레이드 시점이 제한됩니다.

실무에서는 업그레이드 경로를 단순히 한 단계씩 진행하는 것이 아니라, 여러 버전을 건너뛰는 경우가 많습니다. 그러나 WebLogic에서는 2단계 이상 건너뛰기(예: 10.3.6 → 14.1.1)는 사실상 전면 재검증이 필요하며, 기존 설정 및 배포 기술자의 호환성 검증, API 변경에 따른 테스트, 데이터소스 및 JMS 설정의 재구성이 필수적입니다. 또한, WebLogic 전용 기능(Coherence, T3 프로토콜 등)의 유지 여부와 신규 버전에서의 지원 범위도 반드시 확인해야 합니다.

결론적으로, WebLogic 상위 버전 업그레이드는 기존 시스템의 연속성과 벤더 지원을 유지할 수 있지만, 각 단계마다 JDK, Java EE 스펙, 배포 기술자, API 호환성 등 복잡한 기술적 제약이

존재하며, 대규모 코드 수정과 전면 검증이 필요하다는 점을 반드시 고려해야 합니다.

3.1.2 WebLogic 업그레이드 방식의 장점과 한계

WebLogic 상위 버전 업그레이드 방식은 여러 측면에서 장점과 한계를 동시에 가지고 있습니다. 아래 표를 기반으로 각 항목별로 상세히 분석하겠습니다.

항목	장점	한계
벤더 호환성	동일 벤더 내 업그레이드로 기존 설정 일부 재활용 가능	Oracle 인증 매트릭스 내에서만 OS/JDK 선택 가능
기술 지원	Oracle Premier / Extended Support 재개	라이선스 비용 지속 발생(연간 수억 원 규모)
기능 연속성	WebLogic 전용 기능(Coherence, JMS 등) 유지 가능	Jakarta EE 10 지원은 15.1.1(2025년 예정)까지 대기 필요
마이그레이션 범위	WAS 계층만 변경하여 접근 가능	2단계 이상 버전 점프(예: 10.3 → 14.1) 시 사실상 전면 재검증 필요
벤더 종속	–	Oracle 라이선스 정책 변경에 지속적으로 종속

먼저, 벤더 호환성 측면에서는 동일 Oracle 제품군 내에서 업그레이드가 이루어지므로, 기존 시스템의 설정(도메인 구성, 데이터소스, JMS 등)을 일부 재활용할 수 있습니다. 이는 마이그레이션 범위가 WAS 계층에 한정되어 있어, 전체 시스템의 변경 부담을 줄일 수 있다는 장점이 있습니다. 또한, 업그레이드 후 Oracle의 Premier 또는 Extended Support를 다시 받을 수 있으므로, 보안 패치와 기술 지원이 재개되어 실무 운영에서 안정성을 확보할 수 있습니다.

기능 연속성 측면에서는 WebLogic 전용 기능(Coherence, JMS, T3 프로토콜 등)을 그대로 유지할 수 있습니다. 이는 기존 애플리케이션이 WebLogic 독점 API에 의존하고 있을 경우, 기능적 호환성을 확보하는 데 유리합니다. 그러나 Jakarta EE 10 지원은 15.1.1(2025년 예정) 버전까지 기다려야 하며, 최신 Java/Jakarta EE 생태계와의 호환성은 여전히 제한적입니다.

한편, 업그레이드 방식의 한계도 분명합니다. Oracle의 인증 매트릭스 내에서만 OS와 JDK를 선택할 수 있으므로, 최신 Linux 배포판이나 OpenJDK와의 자유로운 조합이 어렵습니다. 또한, 라이선스 비용이 지속적으로 발생하며, Extended Support 단계에서도 연간 수억 원의 비용이 청구됩니다. 이는 비용 대비 가치가 점차 하락하는 구조를 만듭니다.

마이그레이션 범위가 WAS 계층에 한정되어 있지만, 2단계 이상 버전을 건너뛰는 경우(예:

10.3.6 → 14.1.1)에는 사실상 전면 재검증이 필요합니다. 기존 설정, 배포 기술자, API, 데이터소스, JMS 등 모든 구성 요소를 다시 테스트하고 검증해야 하므로, 실무적 난이도가 매우 높아집니다.

마지막으로, 벤더 종속성은 업그레이드 이후에도 계속 유지됩니다. Oracle의 라이선스 정책 변경이나 지원 종료(EOS) 이슈가 발생할 경우, 다시 한 번 대규모 마이그레이션이 필요해질 수 있습니다. 따라서 장기적인 관점에서 기술 부채와 비용 구조를 반드시 검토해야 합니다.

WebLogic 업그레이드 방식의 실제 사례를 살펴보면, 대형 금융기관이나 공공기관에서는 기존 시스템의 안정성과 벤더의 공식 지원을 중시하여 업그레이드 방식을 선택하는 경우가 많습니다. 예를 들어, 대규모 트랜잭션 처리와 복잡한 EJB/JMS 연동이 필요한 환경에서는 WebLogic의 전용 기능과 오랜 운영 경험이 중요한 자산으로 작용합니다. 그러나 이러한 선택은 장기적으로 벤더 종속성과 비용 부담, 그리고 기술 부채 누적이라는 문제로 이어질 수 있습니다. 특히, 최근 클라우드 네이티브 환경으로의 전환이 가속화됨에 따라, WebLogic의 컨테이너 지원 한계와 최신 DevOps 도구와의 통합 제약이 점점 더 뚜렷해지고 있습니다. 따라서 업그레이드 방식의 선택 시에는 단기적 안정성뿐만 아니라, 장기적 기술 전략과 조직의 IT 로드맵을 함께 고려하는 것이 매우 중요합니다. 또한, 업그레이드 프로젝트를 추진할 때에는 사전 PoC(Proof of Concept) 수행, 자동화된 테스트 도입, 그리고 주요 이해관계자와의 긴밀한 협업이 성공적인 마이그레이션의 핵심 요소임을 명심해야 합니다.

3.2 오픈소스 WAS 전환 방식

최근 엔터프라이즈 환경에서는 WebLogic과 같은 상용 WAS를 오픈소스 WAS로 전환하는 사례가 급증하고 있습니다. 이는 라이선스 비용 절감, 벤더 종속성 탈피, 최신 기술 적용, 클라우드 네이티브 환경에 대한 적합성 등 다양한 이유에서 비롯됩니다. 오픈소스 WAS는 Apache Tomcat, WildFly(JBoss), Open Liberty, Apache TomEE 등 다양한 제품이 존재하며, 각 WAS별로 지원하는 Java EE/Jakarta EE 스펙, EJB 지원 여부, JDK 최신 버전 호환성, 컨테이너/Kubernetes 적합성, 상용 지원 옵션, 시장 점유율 등이 상이합니다. 본 절에서는 오픈소스 WAS의 비교와 전환 방식의 장단점, 그리고 실무적 고려사항을 상세히 다룹니다.

3.2.1 대안 오픈소스 WAS 비교

WebLogic 대안으로 검토되는 오픈소스 WAS는 Apache Tomcat, WildFly(JBoss), Open Liberty, Apache TomEE 등이 대표적입니다. 각 WAS의 특징을 비교하면 다음과 같습니다.

비교 항목	Apache Tomcat	WildFly (JBoss)	Open Liberty	Apache TomEE
라이선스	Apache 2.0	LGPL 2.1	EPL 1.0	Apache 2.0
Java EE/Jakarta EE 지원	Servlet/JSP만 (Web Profile 아님)	Full Profile (Jakarta EE 10)	Full/Web Profile (Jakarta EE 10)	Web/Full Profile
EJB 지원	미지원 (OpenEJB 추가 가능)	완전 지원	완전 지원	완전 지원
JDK 17/21 지원	지원 (Tomcat 10.1+)	지원 (WildFly 27+)	지원	지원
컨테이너/K8s 적합성	매우 높음 (경량)	높음	높음	높음
상용 지원 옵션	없음 (3rd party)	Red Hat JBoss EAP	IBM WebSphere Liberty	Tomitribe
시장 점유율 (2025)	66%	14%	~5%	~2%
Spring Boot 내장 서버	기본 내장	미지원	미지원	미지원

Apache Tomcat은 경량 Servlet/JSP 컨테이너로, 시장 점유율이 가장 높으며 Spring Boot 내장 서버로도 사용됩니다. Java EE Full Profile은 지원하지 않지만, 대부분의 현대 애플리케이션이 Spring 기반으로 개발되는 경우 적합합니다. WildFly(JBoss)는 Jakarta EE 10 Full Profile을 지원하며, EJB, JMS, JCA 등 엔터프라이즈 기능을 완전하게 제공합니다. Open Liberty는 IBM이 주도하는 WAS로, Full/Web Profile 지원과 뛰어난 컨테이너/Kubernetes 적합성을 갖추고 있습니다. Apache TomEE는 Tomcat 기반에 EJB, JPA, CDI 등 Java EE 기능을 추가한 WAS로, 경량성과 확장성을 동시에 제공합니다.

JDK 17/21 등 최신 Java 버전 지원은 모든 WAS에서 가능하며, 컨테이너/Kubernetes 환경에 최적화된 구조를 갖추고 있습니다. 상용 지원 옵션은 WildFly(JBoss EAP), Open Liberty(IBM WebSphere Liberty), TomEE(Tomitribe) 등에서 제공되므로, 엔터프라이즈 환경에서 SLA를 요구하는 경우에도 대응이 가능합니다.

실무적으로는 애플리케이션의 Java EE/Jakarta EE 기능 사용 수준, EJB/JMS/JCA 의존성, Spring Boot 기반 여부, 상용 지원 요구사항 등을 기준으로 WAS를 선택해야 합니다. 시장

점유율과 커뮤니티 활성도, 기술 지원 범위도 중요한 판단 요소입니다.

오픈소스 WAS의 선택은 단순히 기술적 사양만을 비교하는 것이 아니라, 실제 운영 환경과 조직의 요구사항을 종합적으로 고려해야 합니다. 예를 들어, 대규모 트랜잭션 처리나 복잡한 EJB/JMS 연동이 필요한 경우에는 WildFly나 Open Liberty와 같은 Full Profile 지원 WAS가 적합할 수 있습니다. 반면, 단순한 웹 애플리케이션이나 Spring Boot 기반의 마이크로서비스 아키텍처를 도입하려는 경우에는 Apache Tomcat이나 TomEE가 더 효율적일 수 있습니다. 또한, 컨테이너 기반 배포와 DevOps 자동화, 클라우드 네이티브 환경에 대한 지원 여부도 중요한 선택 기준입니다. 최근에는 Kubernetes 오퍼레이터, Helm 차트, CI/CD 파이프라인과의 통합 등 오픈소스 WAS의 생태계가 급속도로 발전하고 있어, 장기적인 기술 전략 수립 시 이러한 요소들을 반드시 고려해야 합니다. 실제로 국내외 대기업 및 공공기관에서도 오픈소스 WAS로의 전환을 통해 라이선스 비용을 절감하고, 최신 기술 도입 및 인력 수급의 유연성을 확보한 사례가 증가하고 있습니다. 따라서 오픈소스 WAS 선택 시에는 단기적 비용 절감뿐만 아니라, 조직의 IT 로드맵과 미래 지향적 기술 전략을 함께 검토하는 것이 바람직합니다.

3.2.2 오픈소스 WAS 전환의 장점과 한계

오픈소스 WAS 전환은 비용, 기술 표준, 개발 생태계, 클라우드 네이티브, 인력 수급 등 다양한 측면에서 장점과 한계를 가지고 있습니다. 아래 표를 기반으로 각 항목별로 상세히 분석하겠습니다.

항목	장점	한계
비용	라이선스 무료, TCO 대폭 절감	상용 기술 지원 필요 시 별도 계약(Red Hat EAP 등)
기술 표준	최신 Jakarta EE, JDK 17/21 즉시 적용 가능	WebLogic 전용 기능(Coherence, T3 등) 대체 구현 필요
개발 생태계	Spring Boot, MicroProfile 등 현대 프레임워크와 자연스러운 통합	기존 EJB 코드의 대규모 리팩토링 필요
클라우드 네이티브	컨테이너화, Kubernetes 배포에 최적화	기존 운영팀의 학습 곡선
인력 수급	Java/Spring 개발자 풀이 압도적으로 넓음	WebLogic 전용 지식의 전환 비용

비용 측면에서는 오픈소스 WAS가 라이선스 무료로 운영 비용을 대폭 절감할 수 있습니다. 유지보수 비용도 자체 기술 역량에 따라 절감 가능하며, 상용 지원이 필요한 경우 Red Hat JBoss EAP, IBM WebSphere Liberty 등 별도 계약이 가능합니다.

기술 표준 측면에서는 최신 Jakarta EE, JDK 17/21 등 최신 기술을 즉시 적용할 수 있어, 기술 부채 해소와 성능·보안 개선에 유리합니다. 그러나 WebLogic 전용 기능(Coherence, T3 프로토콜 등)은 오픈소스 WAS에서 대체 구현이 필요하며, 일부 기능은 완전한 호환이 어렵습니다.

개발 생태계 측면에서는 Spring Boot, MicroProfile 등 현대 프레임워크와 자연스러운 통합이 가능하며, 개발자 풀이 넓어 신규 인력 채용과 기술 역량 강화에 유리합니다. 반면, 기존 EJB 코드의 대규모 리팩토링이 필요하며, 전환 과정에서 코드 수정과 테스트 부담이 큽니다.

클라우드 네이티브 측면에서는 오픈소스 WAS가 컨테이너화, Kubernetes 배포에 최적화되어 있어, 클라우드 환경으로의 전환이 용이합니다. 그러나 기존 운영팀의 학습 곡선이 존재하며, 새로운 운영 방식에 대한 교육과 적응이 필요합니다.

인력 수급 측면에서는 Java/Spring 개발자 풀이 압도적으로 넓어, 유지보수와 신규 개발에 유리합니다. 그러나 WebLogic 전용 지식의 전환 비용이 발생하며, 기존 시스템의 기술적 특성을 이해하고 전환하는 데 추가 리소스가 요구됩니다.

실제 오픈소스 WAS 전환 사례를 살펴보면, 대형 유통사나 금융기관에서 라이선스 비용 절감과 클라우드 전환을 위해 Tomcat 또는 WildFly로의 마이그레이션을 추진한 경우가 많습니다. 이 과정에서 기존 EJB 기반 애플리케이션을 Spring Boot 기반으로 리팩토링하거나, JMS 연동 방식을 ActiveMQ 등 오픈소스 메시징 솔루션으로 대체하는 전략이 활용됩니다. 또한, CI/CD 자동화와 컨테이너 오케스트레이션(Kubernetes) 도입을 통해 배포 효율성과 운영 안정성을 동시에 확보하는 사례가 늘고 있습니다. 반면, WebLogic 전용 기능에 대한 대체 구현이 어렵거나, 대규모 레거시 코드의 리팩토링 부담이 큰 경우에는 단계적 전환 또는 하이브리드 운영 방식을 선택하는 경우도 있습니다. 오픈소스 WAS 전환은 단순한 기술 교체가 아니라, 조직의 개발·운영 문화와 IT 거버넌스까지 변화시키는 계기가 될 수 있으므로, 전환 전략 수립 시 충분한 사전 분석과 파일럿 프로젝트, 그리고 조직 내 기술 역량 강화 방안이 반드시 병행되어야 합니다. 이러한 준비 과정을 통해 오픈소스 WAS 전환의 장점을 극대화하고, 잠재적 한계를 효과적으로 관리할 수 있습니다.

3.3 두 방식의 종합 비교와 판단 기준

WebLogic 상위 버전 업그레이드와 오픈소스 WAS 전환은 각각의 장단점과 실무적 고려사항이 존재합니다. 본 절에서는 두 방식의 종합 비교를 위한 의사결정 매트릭스와, 하이브리드 전략(단

계적 전환 접근법)을 제시하여, IT 의사결정권자가 최적의 마이그레이션 방향을 선택할 수 있도록 안내합니다.

3.3.1 의사결정 매트릭스

마이그레이션 전략을 결정할 때는 비용, 기간, 기술 부채 해소, 향후 유연성, 인력 수급, 클라우드 전환 용이성의 6개 축을 기준으로 비교해야 합니다. 아래 매트릭스는 각 축에 대해 WebLogic 업그레이드와 오픈소스 전환의 상대적 점수(상/중/하)를 부여한 예시입니다.

비교 축	WebLogic 업그레이드	오픈소스 WAS 전환
비용	하 (라이선스 비용 지속)	상 (라이선스 무료, TCO 절감)
기간	하 (대규모 코드 리팩토링 필요)	하 (대규모 코드 리팩토링 필요)
기술 부채 해소	하 (일부만 해소, 최신 EE/JDK 한계)	상 (최신 EE/JDK 즉시 적용)
향후 유연성	하 (벤더 종속 지속)	상 (기술·벤더 유연성 확보)
인력 수급	하 (WebLogic 경험자 부족)	상 (Java/Spring 개발자 풍부)
클라우드 전환 용이성	중 (WebLogic Kubernetes 지원 제한)	상 (컨테이너/K8s 최적화)

이 매트릭스를 기반으로, 다음과 같은 판단 기준을 적용할 수 있습니다.

- **비용 중심**: 예산 절감이 최우선이라면 오픈소스 WAS 전환이 적합합니다.
- **기간 중심**: 단기간 내 안정성 확보가 필요하다면 WebLogic 업그레이드가 유리할 수 있지만, 애플리케이션과 인프라의 대규모 전환은 동일하게 필수 적입니다.
- **기술 부채 해소**: 최신 EE/JDK, 클라우드 네이티브 환경을 목표로 한다면 오픈소스 WAS 전환이 바람직합니다.
- **인력 수급**: WebLogic 경험자 채용이 어렵고, Java/Spring 개발자 풀이 넓은 경우 오픈소스 WAS 전환이 유리합니다.
- **유연성/확장성**: 장기적 기술 유연성과 벤더 종속 탈피를 원한다면 오픈소스 WAS가 적합합니다.

실무에서는 각 축별로 조직의 우선순위와 리소스 상황을 고려하여, 종합적인 판단을 내려야 합니다.

의사결정 매트릭스는 단순한 점수 비교를 넘어, 실제 프로젝트 추진 시 발생할 수 있는 리스크와 기회 요소를 체계적으로 분석하는 데 중요한 역할을 합니다. 예를 들어, 대규모 금융기관에서는 규제 준수와 안정성 확보가 최우선이기 때문에, 중기적으로는 WebLogic 업그레이드를 선택하되, 중장기적으로는 오픈소스 WAS 전환을 준비하는 하이브리드 전략을 채택하는 경우가 많습니다. 반면, 스타트업이나 IT 서비스 기업처럼 민첩성과 비용 효율성이 중요한 조직에서는 초기부터 오픈소스 WAS를 도입하여 최신 기술을 적극적으로 활용하는 사례가 두드러집니다. 또한, 클라우드 전환을 목표로 하는 조직에서는 컨테이너/Kubernetes 환경에 최적화된 오픈소스 WAS의 도입이 필수적이며, DevOps, IaC(Infrastructure as Code), 자동화 테스트 등 현대적 IT 운영 방식과의 연계성도 중요한 고려 사항입니다. 따라서 의사결정 매트릭스는 조직의 비즈니스 목표, IT 전략, 인력 구성, 그리고 외부 환경 변화까지 종합적으로 반영하여, 최적의 마이그레이션 경로를 도출하는 데 활용되어야 합니다. 실제로, 성공적인 마이그레이션 프로젝트에서는 매트릭스 기반의 사전 평가와 이해관계자 협의, 그리고 단계별 실행 계획 수립이 프로젝트 리스크를 최소화하는 핵심 요소로 작용합니다.

3.3.2 하이브리드 전략: 단계적 전환 접근법

마이그레이션 전략은 단순히 업그레이드 또는 전환 중 하나를 선택하는 것이 아니라, 상황에 따라 하이브리드 전략을 적용하는 것이 실무적으로 더욱 효과적일 수 있습니다. 특히 WebLogic 7, 8, 9 등 구버전 환경에서는 상위 WebLogic 버전 업그레이드보다 오픈소스 WAS 전환이 비용 및 기술 부채 측면에서 더 유리합니다. 구버전 WebLogic은 JDK, OS, Java EE 표준, 보안 패치 등 모든 계층에서 단종 위험이 누적되어 있으므로, 전면적인 기술 스택 전환이 필요합니다.

하이브리드 전략은 다음과 같은 단계적 접근법을 권장합니다.

1. **1단계: WebLogic 업그레이드(단기 위험 해소)**
    - 즉각적인 보안 위험(CVE, PCI DSS 등)을 해소하기 위해 WebLogic 상위 버전으로 업그레이드합니다.
    - 이 과정에서 최소한의 코드 수정과 설정 변경으로 운영 안정성을 확보합니다.
2. **2단계: 신규 개발은 오픈소스 WAS/Spring Boot 기반 진행**

- 신규 프로젝트 또는 신규 기능 개발은 Spring Boot, WildFly, Open Liberty 등 오픈소스 WAS 기반으로 진행합니다.
- 최신 Jakarta EE, JDK, 클라우드 네이티브 환경을 적극 활용하여 기술 부채를 최소화합니다.

3. **3단계: 기존 애플리케이션 점진적 이전(중장기 전략)**

- 기존 WebLogic 애플리케이션은 단계적으로 오픈소스 WAS로 이전합니다.
- 파일럿 프로젝트, 우선순위별 분류(Green/Yellow/Red), 코드 리팩토링, 표준 API 전환, CI/CD 파이프라인 구축 등을 통해 점진적 마이그레이션을 수행합니다.

이 전략은 단기적으로 운영 위험을 해소하면서, 중장기적으로 기술 부채를 해소하고 비용 구조를 개선할 수 있습니다. 또한, 조직 내 기술 역량 강화와 개발 생태계 현대화, 클라우드 전환 기반을 동시에 마련할 수 있습니다.

하이브리드 전략의 실제 적용 사례를 살펴보면, 대형 제조업체나 공공기관에서는 기존 WebLogic 기반의 핵심 시스템을 단기적으로 업그레이드하여 운영 리스크를 최소화한 뒤, 신규 서비스나 마이크로서비스 아키텍처 도입 시 오픈소스 WAS와 클라우드 네이티브 기술을 적극 활용하는 방식을 채택하고 있습니다. 이러한 접근법은 기존 시스템의 안정성과 신규 기술 도입의 유연성을 동시에 확보할 수 있다는 장점이 있습니다. 또한, 단계적 전환 과정에서 애플리케이션의 표준화, 코드 리팩토링, DevOps 도입, 자동화 테스트 환경 구축 등 IT 역량 강화를 병행할 수 있어, 장기적으로 조직의 기술 경쟁력을 크게 높일 수 있습니다. 하이브리드 전략은 단순히 기술적 선택의 문제가 아니라, 조직의 변화 관리, 인력 재교육, 그리고 IT 거버넌스 체계의 혁신까지 포함하는 종합적 변화 관리 전략임을 명확히 인식해야 합니다. 성공적인 하이브리드 마이그레이션을 위해서는 경영진의 지원, 명확한 로드맵 수립, 그리고 단계별 성과 측정과 피드백 체계가 반드시 필요합니다.

결론적으로, WebLogic 업그레이드와 오픈소스 WAS 전환은 각각의 장단점과 실무적 제약이 존재하므로, 조직의 목표와 리소스 상황에 맞춰 하이브리드 전략을 적용하는 것이 가장 효과적인 마이그레이션 방식임을 강조합니다.

제4장. WebLogic 마이그레이션 실행 시 핵심 검토 항목과 난제

4.1 마이그레이션 사전 분석에서 반드시 확인할 항목

WebLogic Server 환경에서 오픈소스 WAS 또는 상위 버전으로의 마이그레이션을 계획할 때, 사전 분석 단계는 성공적인 전환의 핵심입니다. 이 단계에서는 기존 시스템의 기술적 의존성, 아키텍처 구조, 트랜잭션 처리 방식 등 다양한 요소를 면밀히 점검해야 합니다. WebLogic은 오랜 기간 엔터프라이즈 환경에서 사용되어 왔으며, 그만큼 전용 API, 배포 기술자, 프로토콜, 보안 설정 등 독자적인 기능에 대한 의존성이 깊게 자리잡고 있습니다. 이러한 요소들은 마이그레이션 시 표준 Java EE/Jakarta EE 또는 Spring 기반 환경으로의 전환을 어렵게 만드는 주요 장애물입니다. 또한, 애플리케이션의 레이어별 의존성 분석, 트랜잭션 처리 방식의 차이, 외부 연동 구조에 대한 이해는 전환 범위와 난이도를 결정하는 데 중요한 역할을 합니다. 최근에는 정적 분석 도구와 자동화 진단 툴(RHAMT, IBM Transformation Advisor 등)을 활용하여, 코드 내 숨겨진 WebLogic 종속성을 빠르게 식별하고, 작업 난이도(Level of Effort)를 정량적으로 평가하는 접근이 보편화되고 있습니다. 이처럼 사전 분석 단계에서의 철저한 준비는 마이그레이션 후 발생할 수 있는 기능 누락, 성능 저하, 보안 취약점 등 예기치 못한 문제를 예방하는 데 필수적입니다.

4.1.1 WebLogic 전용 API 및 기능 의존성 식별

WebLogic Server 환경에서 마이그레이션을 준비할 때 가장 먼저 수행해야 할 작업은 WebLogic 전용 API 및 기능에 대한 의존성 식별입니다. WebLogic은 BEA 시절부터 Oracle 인수 이후까지 독자적인 기능과 API를 지속적으로 확장해왔기 때문에, 애플리케이션 코드와 설정 파일 곳곳에 WebLogic 특화 요소가 자리잡고 있습니다. 이러한 요소들은 표준 Java EE/Jakarta EE 또는 오픈소스 WAS 환경에서는 지원되지 않거나, 완전히 다른 방식으로 구현되어야 하므로, 전환 작업의 난이도를 높이는 주요 원인입니다.

실무적으로 점검해야 할 WebLogic 전용 의존성 체크리스트는 다음과 같습니다.

- **패키지 의존성**: **`com.bea.*`**, **`com.weblogic.*`**, **`weblogic.*`** 패키지의 사용 여부를 코드

전체에서 검색해야 합니다. 이 패키지들은 WebLogic 전용 API를 제공하며, 예를 들어 WebLogic의 JMX, JMS, Work Manager, ApplicationLifecycleListener 등은 타 WAS에서 직접 지원하지 않습니다.

- **배포 기술자**: WebLogic은 표준 **`web.xml`** 외에도 **`weblogic.xml`**, **`weblogic-ejb-jar.xml`**, **`weblogic-cmp-rdbms-jar.xml`**, **`weblogic-application.xml`** 등 전용 배포 기술자를 사용합니다. 이 파일들은 WebLogic의 클러스터링, 데이터소스, 보안, EJB 설정 등 다양한 기능을 세밀하게 제어하는 역할을 하며, 오픈소스 WAS에서는 별도의 설정 파일 또는 표준 API로 대체해야 합니다.
- **ApplicationLifecycleListener**: WebLogic의 ApplicationLifecycleListener는 애플리케이션의 시작과 종료 이벤트를 감지하여 특정 작업을 수행하는 데 사용됩니다. 표준 Java EE에서는 **`@Startup`**, **`@PostConstruct`**, **`@PreDestroy`** 등으로 기능을 대체할 수 있지만, 코드 구조의 변경이 필요합니다.
- **T3 프로토콜 의존성**: WebLogic만의 독자적인 T3 프로토콜은 클러스터링, JNDI, RMI 통신 등에 사용됩니다. 오픈소스 WAS에서는 T3를 지원하지 않으므로, JNDI Lookup, 클러스터링, 원격 호출 방식의 전면적인 변경이 필요합니다.
- **JMS, JDBC DataSource, Work Manager 설정**: WebLogic의 JMS, JDBC DataSource, Work Manager 등은 전용 API와 설정 파일로 관리됩니다. 오픈소스 WAS에서는 표준 JMS, JDBC, Thread Pool 관리 방식으로 전환해야 하며, 설정 파일 매핑이 필요합니다.
- **클래스로더 정책**: WebLogic은 **`prefer-web-inf-classes`**, **`prefer-application-packages`** 등 독자적인 클래스로더 정책을 지원합니다. 이는 라이브러리 충돌 방지와 우선순위 제어에 쓰이지만, 오픈소스 WAS에서는 별도의 설정 또는 구조적 변경이 필요합니다.
- **보안 Realm/Provider**: WebLogic의 보안 Realm, Provider 설정은 인증, 권한 관리에 밀접하게 연관되어 있습니다. 오픈소스 WAS에서는 JAAS, Spring Security, 또는 WAS별 보안 설정으로 대체해야 하며, 보안 정책의 재설계가 필요할 수 있습니다.

이러한 체크리스트를 기반으로, 코드와 설정 파일을 정적 분석하여 WebLogic 전용 요소를 모두 식별하고, 각 항목별로 전환 난이도와 대체 방안을 마련하는 것이 마이그레이션 성공의 첫걸음입니다. 예를 들어, WebLogic T3 JNDI 바인딩이나 InitialContextFactory 변경은 복잡도가

높으므로, 문서화된 솔루션을 참고하여 단계적으로 전환해야 합니다. 반면, 단순한 1:1 라이브러리 변경이 가능한 경우에는 빠르게 수정이 가능합니다. 실제로 AMT, RHAMT 등 진단 도구에서는 각 항목별로 스토리 포인트(난이도)를 부여하여 작업량을 산정하는 방식이 활용됩니다.

WebLogic의 전용 API 및 기능 의존성 식별 과정에서 주의해야 할 점은, 단순히 코드 내 import 문을 검색하는 것에 그치지 않고, 런타임 시점에 동적으로 로딩되는 클래스, 리플렉션을 통한 접근, 외부 설정 파일에서 참조되는 리소스까지 모두 포함해야 한다는 것입니다. 예를 들어, 일부 배치 작업이나 스케줄러는 WebLogic의 TimerService 또는 Work Manager에 의존하고 있을 수 있으며, 이는 오픈소스 WAS에서는 Quartz Scheduler, Spring Task 등으로 대체가 필요합니다. 또한, WebLogic의 보안 Provider는 LDAP, SAML, Kerberos 등 다양한 외부 인증 시스템과 연동되어 있을 수 있으므로, 이 부분 역시 표준 보안 프레임워크로의 전환 방안을 마련해야 합니다. 실제로 대형 금융권 프로젝트에서는, WebLogic의 T3 프로토콜을 사용하는 내부 시스템 간 연동을 REST API 또는 gRPC 기반으로 전환하면서, 전체 아키텍처의 통신 방식이 근본적으로 변경된 사례가 있습니다. 이처럼 WebLogic 전용 요소의 식별과 대체는 단순한 기술적 작업을 넘어, 시스템 전반의 설계와 운영 방식에 영향을 미치므로, 충분한 사전 검토와 전문가의 참여가 필수적입니다.

4.1.2 애플리케이션 아키텍처 및 의존성 맵핑

마이그레이션 대상 애플리케이션의 아키텍처와 의존성 구조를 체계적으로 분석하는 것은 전환 범위와 난이도를 결정하는 데 있어 매우 중요한 단계입니다. WebLogic 환경에서 개발된 애플리케이션은 일반적으로 프레젠테이션, 비즈니스 로직, 데이터 액세스, 외부 연동 등 여러 레이어로 구성되어 있으며, 각 레이어마다 WebLogic 전용 API나 설정에 대한 의존성이 존재할 수 있습니다.

분석 방법으로는 다음과 같은 절차가 권장됩니다.

1. **레이어별 기술 스택 분석**: 프레젠테이션 레이어에서는 JSP/Servlet, JSF, Spring MVC 등 사용 여부를 확인하고, 비즈니스 로직 레이어에서는 EJB, CDI, Spring Bean 등 기술의 사용 현황을 파악합니다. 데이터 액세스 레이어에서는 JDBC, JPA, Hibernate, MyBatis 등 라이브러리의 버전과 호환성을 점검하며, 외부 연동 레이어에서는 SOAP/REST, JMS, JNDI, RMI 등 연동 방식과 API 의존성을 분석합니다.
2. **의존성 트리 분석**: Maven 또는 Gradle 의존성 트리를 활용하여, 애플리케이션이 참조하는

모든 라이브러리와 그 트랜지티브(간접) 의존성을 시각화합니다. 이를 통해 WebLogic 내장 라이브러리와 애플리케이션 라이브러리 간의 중복, 버전 불일치, 보안 취약점 전파 구조를 식별할 수 있습니다.

3. **정적 분석 도구 활용**: Red Hat Migration Toolkit for Applications(RHAMT/MTA), IBM Transformation Advisor 등 전문 도구를 사용하여, 코드 내 WebLogic 전용 API 사용량, 배포 기술자 의존성, 트랜잭션 처리 방식, JNDI Lookup, JMS 설정 등 마이그레이션 난이도가 높은 영역을 자동으로 리포트 생성합니다. 이 도구들은 각 변경 항목별로 난이도(Level of Effort)를 산정하고, 필수/선택 변경 여부를 구분하여 우선순위를 제시합니다.
4. **외부 연동 영향도 평가**: 데이터베이스, 외부 API, 결제 시스템, SaaS 등과의 연동 구조를 분석하여, WebLogic 환경에서만 동작하는 설정이나 프로토콜(T3 등)이 있는지 점검합니다. 외부 시스템과의 호환성 문제는 마이그레이션 후 장애로 이어질 수 있으므로, 사전 검증이 필요합니다.

실제 사례에서는, 여러 레이어에 걸쳐 WebLogic 전용 API가 혼재되어 있는 경우, 전환 작업이 단순 코드 변경을 넘어 아키텍처 재설계 수준으로 확대되는 경우가 많습니다. 예를 들어, EJB 기반 비즈니스 로직을 Spring Bean으로 전환하거나, WebLogic JMS를 표준 JMS로 변경할 때, 트랜잭션 처리 방식과 메시지 전달 구조의 근본적 수정이 필요합니다. 또한, 라이브러리 버전 불일치로 인한 NoSuchMethodError, ClassNotFoundException 등 런타임 오류가 빈번하게 발생할 수 있으므로, 의존성 트리 분석과 테스트를 반복적으로 수행해야 합니다.

이처럼 애플리케이션 아키텍처 및 의존성 맵핑은 마이그레이션 범위, 작업량, 리소스 배분, 일정 산정의 근거가 되며, 전환 전략 수립의 핵심 자료로 활용됩니다.

애플리케이션 아키텍처 및 의존성 맵핑 과정에서는, 단순히 기술 스택과 라이브러리 목록을 나열하는 것에 그치지 않고, 각 레이어 간의 상호작용과 데이터 흐름, 그리고 외부 시스템과의 연계 방식까지 심층적으로 분석해야 합니다. 예를 들어, 프레젠테이션 레이어에서 WebLogic의 JSP 태그 라이브러리나 JSF 컴포넌트를 사용하는 경우, 오픈소스 WAS로 전환 시 호환성 문제가 발생할 수 있으므로, 표준 태그 라이브러리나 최신 JSF 버전으로의 업그레이드가 필요합니다. 또한, 비즈니스 로직 레이어에서 EJB를 사용하고 있다면, 해당 로직을 Spring Bean이나 CDI로 전환할 때 트랜잭션 경계, 보안 적용 방식, 의존성 주입 방식이 달라지므로, 코드 구조 자체를 재설계해야 할 수 있습니다. 데이터 액세스 레이어에서는 기존에 사용하던 JDBC 드라이버가 오픈소스 WAS에서

지원되지 않거나, JPA 매핑 방식이 달라질 수 있으므로, 데이터베이스 연결 설정과 쿼리 최적화 방안도 함께 검토해야 합니다. 외부 연동 레이어에서는 WebLogic의 T3, RMI, JMS 등 독자적인 프로토콜을 REST, gRPC, 표준 JMS 등으로 변경할 때, 통신 보안, 메시지 포맷, 장애 복구 정책 등도 함께 재설계해야 합니다. 실제로 대규모 공공기관 마이그레이션 프로젝트에서는, 의존성 트리 분석 결과 수십 개의 라이브러리 충돌과 수백 건의 코드 수정이 필요하다는 리포트가 도출된 바 있으며, 이를 기반으로 단계별 전환 계획과 테스트 전략이 수립되었습니다. 이처럼 체계적인 아키텍처 및 의존성 맵핑은 마이그레이션의 성공 확률을 높이고, 예기치 못한 장애를 사전에 예방하는 데 결정적인 역할을 합니다.

4.1.3 트랜잭션 관리 방식 전환

WebLogic Server는 JTA(Java Transaction API) 기반의 컨테이너 관리 트랜잭션(Container Managed Transaction, CMT)을 지원하며, XA(2-Phase Commit) 트랜잭션을 통한 분산 트랜잭션 처리에 강점을 가지고 있습니다. 그러나 오픈소스 WAS나 Spring Framework 기반 환경으로 전환할 때에는 트랜잭션 관리 방식의 구조적 차이와 지원 수준을 반드시 확인해야 합니다.

WebLogic의 CMT는 EJB, JMS, JDBC 등 다양한 리소스에 대한 트랜잭션을 자동으로 관리하며, XA 트랜잭션을 통해 여러 데이터소스(예: DB, MQ 등)에 대한 일관된 커밋/롤백을 보장합니다. 반면, 오픈소스 WAS에서는 CMT를 지원하는 경우도 있지만, Spring Framework에서는 대부분 프로그래밍 방식(Programmatic Transaction) 또는 선언적 방식(Declarative Transaction)을 사용합니다. Spring의 @Transactional 어노테이션은 단일 데이터소스에 대한 트랜잭션 관리에 적합하며, XA 트랜잭션을 사용하려면 Atomikos, Narayana, Bitronix 등 별도의 트랜잭션 매니저를 연동해야 합니다.

마이그레이션 시 주의할 점은 다음과 같습니다.

- **XA 트랜잭션 지원 수준 확인**: 대상 WAS가 XA 트랜잭션(2-Phase Commit)을 지원하는지, 어떤 트랜잭션 매니저를 사용하는지 확인해야 합니다. 예를 들어, WildFly(JBoss)는 Narayana 트랜잭션 매니저를 기본 제공하며, Open Liberty는 JTA 기반 XA 트랜잭션을 지원합니다. Spring Boot에서는 외부 트랜잭션 매니저 연동이 필요합니다.
- **트랜잭션 경계 변경**: WebLogic의 CMT는 EJB 메서드 단위로 트랜잭션 경계를 정의하지만, Spring에서는 @Transactional 어노테이션으로 클래스/메서드 단위 트랜잭션을 지

정합니다. 트랜잭션 전파(Propagation), 격리 수준(Isolation), 롤백 정책 등 세부 설정을 재검토해야 합니다.

- **JNDI 기반 트랜잭션 리소스 전환**: WebLogic에서는 JNDI를 통해 트랜잭션 리소스를 참조하지만, Spring에서는 DI(Dependency Injection) 또는 CDI를 사용합니다. JNDI Lookup 코드를 제거하고, 표준 방식으로 리소스를 주입해야 합니다.
- **분산 트랜잭션의 한계**: 오픈소스 WAS 환경에서는 XA 트랜잭션의 성능 저하, 장애 발생 시 복구 복잡성, 커넥션 풀 관리 방식의 차이 등 실무적 한계가 존재합니다. 가능하다면 단일 데이터소스 트랜잭션으로 아키텍처를 단순화하는 것이 권장됩니다.

실무에서는, 트랜잭션 관리 방식의 변경으로 인해 데이터 일관성, 성능, 장애 복구 등 다양한 이슈가 발생할 수 있으므로, 마이그레이션 후 충분한 테스트와 검증이 필수적입니다. 특히 XA 트랜잭션을 사용하는 대형 금융, 결제, 물류 시스템에서는 트랜잭션 매니저의 설정, 리소스 등록, 장애 복구 정책 등을 세밀하게 설계해야 합니다.

트랜잭션 관리 방식 전환 과정에서 추가적으로 고려해야 할 사항은, 트랜잭션 롤백 정책 및 예외 처리 방식의 차이입니다. WebLogic 환경에서는 시스템 예외 발생 시 자동으로 트랜잭션이 롤백되지만, Spring에서는 예외의 종류(Checked/Unchecked)에 따라 롤백 동작이 달라질 수 있으므로, @Transactional 어노테이션의 rollbackFor 속성 등을 명확히 지정해야 합니다. 또한, WebLogic의 트랜잭션 타임아웃, 리트라이 정책, 트랜잭션 로그 저장 방식 등은 오픈소스 WAS나 Spring 환경에서 별도의 설정이 필요하며, 장애 발생 시 트랜잭션 복구 시나리오도 사전에 테스트해야 합니다. 실제로 대형 커머스 시스템의 마이그레이션 사례에서는, WebLogic의 XA 트랜잭션을 Atomikos로 전환하는 과정에서 트랜잭션 로그 손상, 중복 커밋, 데이터 불일치 등의 문제가 발생하였으며, 이를 해결하기 위해 트랜잭션 로그 백업, 장애 복구 자동화 스크립트, 커넥션 풀 모니터링 도구를 추가로 도입한 바 있습니다. 이처럼 트랜잭션 관리 방식의 전환은 단순한 코드 변경을 넘어, 시스템의 신뢰성과 안정성에 직결되는 핵심 작업이므로, 충분한 사전 검토와 반복 테스트가 반드시 필요합니다.

4.2 마이그레이션 실행 시 가장 어려운 기술 이슈

WebLogic 마이그레이션의 실행 단계에서는 다양한 기술적 난제가 실제로 발생합니다. 특히 EJB 의존 코드의 전환, 전용 설정과 배포 구조 변환, 클래스로더 충돌, 성능 검증 등은 단순 코드 수정만으로 해결되지 않는 복잡한 문제들이며, 실무 경험과 전문 지식이 요구됩니다. 이러한 이슈들은 마이그레이션 작업의 전체 일정과 품질에 직접적인 영향을 미치므로, 사전 분석 단계에서 충분히 대비하고, 실행 단계에서는 반복적인 테스트와 검증을 통해 해결해야 합니다. 또한, 각 기술 이슈는 단일 시스템에 국한되지 않고, 조직 전체의 개발, 운영, 보안 프로세스에 영향을 줄 수 있으므로, 체계적인 대응 전략이 필요합니다.

4.2.1 EJB 의존 코드의 전환

EJB(Enterprise JavaBeans)는 WebLogic 환경에서 비즈니스 로직, 트랜잭션 관리, 메시지 처리 등 다양한 엔터프라이즈 기능을 구현하는 핵심 기술이었습니다. 그러나 최근에는 Spring Framework, CDI, JPA 등으로 대체되는 추세이며, EJB 의존 코드를 오픈소스 WAS 또는 현대적 프레임워크로 전환하는 작업은 마이그레이션에서 가장 난이도가 높은 영역입니다.

구체적으로 전환해야 할 EJB 유형과 전환 방식은 다음과 같습니다.

- **Stateless Session Bean → Spring @Service / CDI @ApplicationScoped**: Stateless Session Bean은 상태를 저장하지 않는 비즈니스 로직 컴포넌트입니다. Spring에서는 @Service 또는 CDI의 @ApplicationScoped로 대체할 수 있습니다. 그러나 WebLogic 환경에서 JNDI Lookup을 통해 EJB를 참조하던 코드는, Spring DI 또는 CDI 방식으로 전면 수정해야 하며, 배포 기술자(weblogic-ejb-jar.xml)도 표준 설정으로 이관해야 합니다.
- **Stateful Session Bean → Spring @SessionScope / CDI @SessionScoped**: Stateful Session Bean은 세션별 상태를 관리하는 컴포넌트입니다. Spring에서는 @SessionScope, CDI에서는 @SessionScoped로 대체할 수 있지만, 상태 관리 방식의 근본적 차이로 인해 설계 변경이 필요합니다. 예를 들어, WebLogic의 세션 복제, failover 정책을 Spring Session 또는 Redis 기반으로 재구현해야 할 수 있습니다.
- **Entity Bean (EJB 2.x) → JPA Entity**: EJB 2.x의 Entity Bean은 데이터베이스와 직접 매핑되는 객체로, EJB 3.x에서는 JPA Entity로 완전히 대체되었습니다. Entity Bean 개념 자

체가 제거되었으므로, 데이터 액세스 레이어를 전면 재작성해야 하며, 기존 CMP(Container Managed Persistence) 설정을 JPA 매핑으로 변환해야 합니다.

- **Message-Driven Bean → JMS Listener / Spring @JmsListener**: Message-Driven Bean(MDB)은 JMS 메시지 처리에 사용됩니다. 오픈소스 WAS에서는 표준 JMS Listener 또는 Spring @JmsListener로 대체할 수 있습니다. WebLogic 전용 설정(weblogic-ejb-jar.xml)을 표준 JMS 설정으로 변환해야 하며, 메시지 큐, 토픽, 연결 팩토리 등 리소스 매핑이 필요합니다.

실제 마이그레이션 사례에서는, EJB 의존 코드의 전환이 단순한 어노테이션 변경을 넘어, 아키텍처 재설계, 트랜잭션 처리 방식 변경, 상태 관리 정책 재구현, 메시지 처리 로직 수정 등 대규모 작업으로 이어집니다. 특히 JNDI Lookup, 배포 기술자, 트랜잭션 경계, 메시지 큐 설정 등 WebLogic 전용 요소가 복합적으로 얽혀 있는 경우, 전환 작업이 매우 복잡해집니다. 또한, EJB 경험 개발자 수급이 어려워진 현실에서, 코드 전환과 검증을 위한 전문 인력 확보가 중요한 과제로 부상하고 있습니다.

EJB 의존 코드 전환의 난이도는 실제로 프로젝트의 규모와 복잡성, 그리고 기존 시스템의 설계 방식에 따라 크게 달라집니다. 예를 들어, 대규모 금융 시스템이나 공공기관의 정보 시스템에서는 수십 개의 EJB 모듈이 서로 복잡하게 얽혀 있고, 각 모듈이 트랜잭션, 보안, 메시지 처리 등 다양한 엔터프라이즈 기능을 담당하고 있습니다. 이러한 시스템에서 EJB를 Spring Bean이나 CDI로 전환할 때는, 단순히 클래스 어노테이션을 변경하는 수준을 넘어, 전체 서비스 아키텍처와 데이터 흐름, 트랜잭션 경계, 예외 처리 로직까지 모두 재설계해야 하는 경우가 많습니다. 또한, WebLogic의 EJB TimerService, JMS 연동, 보안 컨텍스트 전파 등은 Spring이나 오픈소스 WAS에서 직접적으로 지원하지 않으므로, Quartz Scheduler, Spring Security, 표준 JMS Listener 등으로 대체 구현이 필요합니다. 실제로 한 대형 유통사의 마이그레이션 프로젝트에서는, EJB 기반의 주문 처리 시스템을 Spring Boot 기반으로 전환하는 과정에서, 트랜잭션 처리 방식의 차이로 인한 데이터 정합성 문제, 메시지 큐 설정 오류, 세션 상태 관리 이슈 등이 발생하였으며, 이를 해결하기 위해 수차례의 코드 리팩토링과 통합 테스트, 장애 복구 시나리오 검증이 반복되었습니다. 이처럼 EJB 의존 코드의 전환은 단순한 기술적 작업을 넘어, 시스템 전체의 신뢰성과 안정성, 그리고 유지보수성을 좌우하는 핵심 과제이므로, 충분한 사전 분석과 전문가의 참여, 그리고 단계별 검증이 반드시 필요합니다.

4.2.2 WebLogic 전용 설정과 배포 구조 변환

WebLogic Server는 도메인 구성(config.xml), 클러스터 설정, 데이터소스, JMS, 보안 영역 등 다양한 설정을 전용 파일과 방식으로 관리합니다. 마이그레이션 시에는 이러한 설정을 대상 WAS의 표준 파일 또는 방식으로 변환해야 하며, 일부 기능은 1:1 매핑이 불가능하여 대체 구현이 필요합니다.

다음은 주요 설정 항목과 매핑 방식입니다.

WebLogic 설정 항목	대상 WAS 매핑 방식	비고/대안
config.xml	server.xml, standalone.xml, application.yaml	WAS별 설정 파일 구조 상이
weblogic.xml	web.xml, context.xml	표준 설정으로 이관
weblogic-ejb-jar.xml	ejb-jar.xml, CDI 설정	일부 기능은 직접 구현 필요
weblogic-cmp-rdbms-jar.xml	persistence.xml(JPA)	CMP → JPA 매핑 필요
데이터소스 설정	datasource.xml, application.yaml	WAS별 표준 방식 적용
JMS 설정	jms.xml, application.yaml	표준 JMS 설정으로 이관
보안 영역 설정	security.xml, JAAS, Spring Security	정책 재설계 필요
Work Manager	Thread Pool, Executor	직접 구현 또는 대체 라이브러리
Coherence 캐시	Hazelcast, Redis, Ehcache 등	별도 캐시 솔루션 연동 필요

실무에서는, WebLogic의 도메인 기반 클러스터링, Work Manager, Coherence 캐시 등 독자적인 기능은 오픈소스 WAS에서 직접 지원하지 않으므로, 별도의 솔루션(예: Hazelcast, Redis, Spring Task Executor 등)을 연동하거나, 아키텍처를 재설계해야 합니다. 또한, 설정 파일 구조와 배포 방식이 WAS마다 다르므로, 기존 설정을 자동 변환하는 도구를 활용하거나, 수작업으로 매핑 테이블을 작성하여 전환 작업을 수행해야 합니다.

설정 변환 과정에서 발생할 수 있는 주요 이슈는 다음과 같습니다.

- 설정 파일 구조 불일치로 인한 기능 누락
- 클러스터링, 세션 복제, 데이터소스 관리 방식의 차이
- 보안 정책, 인증/권한 관리 방식의 재설계 필요
- JMS, Thread Pool 등 리소스 매핑 오류

이러한 이슈는 마이그레이션 후 기능 검증, 성능 테스트, 장애 복구 시나리오 등을 통해 반복적

으로 점검해야 하며, 매핑이 불가능한 기능은 대체 방안을 마련해야 합니다.

WebLogic 전용 설정과 배포 구조 변환 과정에서는, 단순히 설정 파일을 새로운 포맷으로 변환하는 것에 그치지 않고, 각 설정 항목이 실제로 시스템 동작에 미치는 영향까지 면밀히 분석해야 합니다. 예를 들어, WebLogic의 config.xml은 도메인 전체의 서버, 클러스터, 데이터소스, JMS, 보안 영역 등을 통합적으로 관리하는 반면, Tomcat이나 WildFly 등 오픈소스 WAS는 서버별, 애플리케이션별로 설정 파일이 분리되어 있습니다. 이로 인해, 기존에 하나의 파일에서 관리하던 설정을 여러 파일로 분리하고, 각 파일 간의 참조 관계를 명확히 해야 하는 추가 작업이 필요합니다. 또한, WebLogic의 Work Manager는 애플리케이션별로 세밀한 스레드 풀 정책을 설정할 수 있지만, 오픈소스 WAS에서는 Thread Pool이나 Executor를 전역적으로 관리하는 경우가 많으므로, 동시성 제어 및 리소스 할당 정책을 재설계해야 할 수 있습니다. 보안 영역 역시 WebLogic의 Realm, Provider 설정이 JAAS, Spring Security 등으로 전환될 때, 인증 방식, 권한 맵핑, 세션 관리 정책 등이 달라질 수 있으므로, 기존 보안 정책의 재설계와 테스트가 필수적입니다. 실제로 한 대형 제조사의 마이그레이션 프로젝트에서는, WebLogic의 클러스터링 설정을 Hazelcast 기반 세션 복제로 전환하면서, 세션 동기화 지연, 데이터 일관성 문제, 장애 복구 시나리오 미비 등 다양한 이슈가 발생하였으며, 이를 해결하기 위해 세션 복제 정책을 세분화하고, 장애 발생 시 자동 복구 스크립트를 추가로 개발한 바 있습니다. 이처럼 WebLogic 전용 설정과 배포 구조 변환은 단순한 포맷 변경을 넘어, 시스템 전체의 운영 안정성과 확장성, 그리고 장애 대응력을 좌우하는 핵심 작업이므로, 충분한 사전 분석과 반복 테스트가 반드시 필요합니다.

4.2.3 클래스로더 충돌과 라이브러리 호환성

WebLogic Server는 독자적인 클래스로더 계층 구조를 갖추고 있습니다. System → Domain → Application → Module 순으로 계층화된 클래스로더는, 라이브러리 충돌 방지와 우선순위 제어에 강점을 제공합니다. 특히 **`prefer-web-inf-classes`**, **`prefer-application-packages`** 정책은 애플리케이션의 라이브러리가 WebLogic 내장 라이브러리보다 우선 적용되도록 설정할 수 있습니다.

오픈소스 WAS에서는 클래스로더 구조가 WAS별로 상이하며, Tomcat, WildFly, Open Liberty 등은 표준 Java EE/Jakarta EE 방식 또는 자체 계층 구조를 사용합니다. 마이그레이션 시에는 다음과 같은 문제가 발생할 수 있습니다.

- **라이브러리 충돌**: WebLogic 내장 라이브러리와 애플리케이션 라이브러리 간 버전 불일치로 인한 NoSuchMethodError, ClassNotFoundException 등 런타임 오류가 발생할 수 있습니다.
- **클래스로더 정책 미지원**: 오픈소스 WAS에서는 WebLogic의 prefer-web-inf-classes와 같은 정책을 직접 지원하지 않으므로, 라이브러리 우선순위 제어를 위해 별도의 설정(예: Tomcat의 loader, WildFly의 module.xml 등)을 적용해야 합니다.
- **트랜지티브 의존성 문제**: Maven/Gradle 의존성 트리에서 간접적으로 참조되는 라이브러리가 WAS 내장 라이브러리와 충돌할 수 있으므로, 의존성 트리 분석과 정적 검사(SonarQube 등)를 반복적으로 수행해야 합니다.

실무에서는, 클래스로더 충돌 문제를 해결하기 위해 다음과 같은 대체 방법을 사용합니다.

- WAS별 라이브러리 우선순위 설정(예: Tomcat의 설정, WildFly의 module.xml)
- 내장 라이브러리 제거 또는 버전 업그레이드
- 애플리케이션 라이브러리 경로 조정 및 패키징 방식 변경
- 정적 분석 도구를 활용한 충돌 위험 사전 탐지

이처럼 클래스로더 충돌과 라이브러리 호환성 문제는 마이그레이션 후 빈번하게 발생하는 장애 원인이므로, 사전 분석과 반복 테스트를 통해 체계적으로 해결해야 합니다.

클래스로더 충돌과 라이브러리 호환성 문제는 실제로 마이그레이션 프로젝트에서 가장 예측하기 어려운 장애 중 하나로 꼽힙니다. 예를 들어, WebLogic 환경에서는 특정 라이브러리의 버전이 내장되어 있어 애플리케이션에서 별도로 라이브러리를 추가하지 않아도 정상 동작하지만, 오픈소스 WAS로 전환 시 동일한 라이브러리가 내장되어 있지 않거나, 내장 라이브러리와 애플리케이션에서 사용하는 라이브러리의 버전이 달라 런타임 오류가 발생할 수 있습니다. 또한, WebLogic의 prefer-web-inf-classes 정책은 애플리케이션의 라이브러리가 WAS 내장 라이브러리보다 우선 적용되도록 보장하지만, Tomcat이나 WildFly 등에서는 별도의 설정을 통해 우선순위를 조정해야 하므로, 설정 누락 시 예상치 못한 충돌이 발생할 수 있습니다. 실제로 한 금융권 마이그레이션 프로젝트에서는, Spring Framework와 Jackson 라이브러리의 버전 불일치로 인해 NoSuchMethodError가 빈번하게 발생하였으며, 이를 해결하기 위해 의존성 트리 분석, 라이브러리 버전 통일, module.xml 설정 변경 등 다양한 조치가 필요하였습니다. 또한, 트랜지티브 의존성

으로 인해 간접적으로 참조되는 라이브러리가 WAS 내장 라이브러리와 충돌하는 사례도 많으므로, Maven/Gradle의 dependency:tree 명령어와 SonarQube, OWASP Dependency-Check 등 정적 분석 도구를 활용하여 사전에 충돌 위험을 탐지하고, 필요 시 라이브러리 제외(exclude) 설정이나 패키징 방식을 변경해야 합니다. 이처럼 클래스로더 충돌과 라이브러리 호환성 문제는 단순한 코드 수정만으로 해결되지 않으며, 시스템 전체의 안정성과 신뢰성에 직접적인 영향을 미치므로, 반복적인 테스트와 전문가의 참여가 반드시 필요합니다.

4.2.4 성능 검증과 튜닝 재수행

WebLogic 환경에서 최적화된 JDBC Connection Pool, Thread Pool, JVM 옵션 등은 오픈소스 WAS로 마이그레이션할 때 그대로 적용되지 않습니다. 각 WAS는 리소스 관리 방식, 세션 복제, 커넥션 풀 관리, JVM 튜닝 방식이 다르기 때문에, 마이그레이션 후 성능 기준선(baseline)을 재설정하고, 부하 테스트를 반드시 수행해야 합니다.

주요 성능 검증 및 튜닝 항목은 다음과 같습니다.

- **JDBC Connection Pool**: WebLogic의 커넥션 풀 설정은 대상 WAS의 표준 방식으로 재구성해야 하며, 커넥션 풀 크기, 타임아웃, 장애 복구 정책 등을 재설계해야 합니다.
- **Thread Pool**: WebLogic의 Work Manager, Thread Pool 설정은 오픈소스 WAS의 Executor, Thread Pool 설정으로 대체해야 하며, 동시성, 스레드 관리 방식의 차이를 고려해야 합니다.
- **JVM 옵션**: WebLogic에서 사용하던 JVM 옵션(GC, Heap, PermGen 등)은 대상 WAS의 권장 옵션으로 재설정해야 하며, JDK 버전별 최적화 방안을 적용해야 합니다.
- **세션 복제 방식**: WebLogic의 도메인 기반 세션 복제는 오픈소스 WAS에서는 Redis, Hazelcast 등 외부 캐시 솔루션을 연동하거나, WAS별 세션 복제 방식을 적용해야 합니다.
- **성능 기준선 재설정**: 마이그레이션 후 부하 테스트(JMeter, Gatling 등)를 통해 성능 기준선을 재설정하고, 병목 구간을 분석하여 튜닝을 반복해야 합니다.

실무에서는, 성능 검증과 튜닝을 반복적으로 수행하여, 마이그레이션 후 시스템이 기존 수준 이상의 성능을 발휘할 수 있도록 최적화합니다. 특히 대규모 트랜잭션, 동시 접속, 메시지 처리 등 엔터프라이즈 환경에서는 성능 저하가 비즈니스 장애로 이어질 수 있으므로, 사전 대비와 체계적인 검증이 필수적입니다.

성능 검증과 튜닝 재수행 과정에서는, 단순히 기존 WebLogic 환경의 설정을 복사하는 것이 아니라, 새로운 WAS의 특성과 권장 설정을 충분히 이해하고, 실제 트래픽 패턴과 비즈니스 요구사항에 맞게 최적화해야 합니다. 예를 들어, Tomcat이나 WildFly의 커넥션 풀은 HikariCP, Apache DBCP 등 다양한 구현체를 사용할 수 있으므로, 각 구현체의 성능 특성, 장애 복구 정책, 모니터링 방식 등을 비교 분석하여 최적의 설정을 적용해야 합니다. Thread Pool 역시 WebLogic의 Work Manager와는 동작 방식이 다르므로, WAS별 Executor 설정, 동시성 제어 정책, 스레드 수 조정 방식 등을 면밀히 검토해야 합니다. JVM 옵션의 경우, WebLogic에서 사용하던 GC, Heap, PermGen 설정이 JDK 8 이상에서는 의미가 없거나, 새로운 GC 정책(G1GC, ZGC 등)을 적용해야 할 수 있으므로, JDK 버전별 권장 옵션을 적용하고, GC 로그 분석, Heap Dump 분석 등을 통해 메모리 누수나 병목 구간을 사전에 탐지해야 합니다. 세션 복제 방식도 WebLogic의 도메인 기반 세션 복제와는 달리, Redis, Hazelcast 등 외부 캐시 솔루션을 연동하거나, WAS별 세션 클러스터링 기능을 활용해야 하므로, 세션 동기화 지연, 데이터 일관성, 장애 복구 정책 등을 충분히 테스트해야 합니다. 실제로 한 대형 커머스 사이트의 마이그레이션 프로젝트에서는, WebLogic에서 Tomcat+Redis 환경으로 전환한 후 초기에는 세션 동기화 지연과 커넥션 풀 고갈로 인한 성능 저하가 발생하였으며, 이를 해결하기 위해 커넥션 풀 크기 조정, Redis 클러스터 구성, JVM GC 튜닝, 부하 테스트 자동화 등 다양한 튜닝 작업이 반복적으로 수행되었습니다. 이처럼 성능 검증과 튜닝 재수행은 마이그레이션의 마지막 단계에서 시스템의 안정성과 확장성을 보장하는 핵심 작업이므로, 충분한 리소스와 전문가의 참여, 그리고 반복적인 테스트와 모니터링이 반드시 필요합니다.

제5장. 오픈소스 WAS 전환 실행 가이드

5.1 전환 대상 WAS 선정 기준

오픈소스 WAS(Web Application Server)로의 전환은 단순히 소프트웨어를 교체하는 것이 아니라, 기존 시스템의 아키텍처, 개발 프로세스, 운영 방식 전반에 걸쳐 변화가 요구되는 전략적 결정입니다. WebLogic과 같은 상용 WAS 환경에서 오픈소스 WAS로의 전환을 고려할 때, 가장 먼저

검토해야 할 것은 애플리케이션이 요구하는 Java EE/Jakarta EE 기능 수준, 기술 지원 필요성, 그리고 조직의 운영 역량입니다. 각 WAS는 지원하는 표준, 기능, 라이선스, 기술 지원 범위가 다르므로, 전환 대상 WAS 선정 과정에서 애플리케이션의 구조와 요구사항을 면밀히 분석해야 합니다. 특히, Servlet/JSP만 사용하는 경량 웹 애플리케이션과 EJB, JMS, JCA 등 엔터프라이즈 기능을 사용하는 복잡한 시스템은 요구하는 WAS의 특성이 크게 다르며, 상용 기술 지원이 필요한 경우 추가적인 비용과 SLA(Service Level Agreement) 조건도 함께 검토해야 합니다. 본 절에서는 애플리케이션 유형별 WAS 선택 가이드와 상용 기술 지원 옵션 비교를 통해, 실무적으로 적합한 WAS 선정 기준을 제시합니다.

5.1.1 애플리케이션 유형별 적합한 WAS 선택

애플리케이션의 Java EE 기능 사용 수준에 따라 적합한 WAS를 선택하는 것은 마이그레이션 성공의 핵심입니다. WebLogic에서 오픈소스 WAS로 전환할 때, 다음과 같은 기준을 적용할 수 있습니다.

첫째, Servlet/JSP만 사용하는 경량 웹 애플리케이션은 Apache Tomcat 또는 Spring Boot 내장 Tomcat이 최적의 선택입니다. Tomcat은 Servlet 4.0, JSP 2.3 등 웹 프로파일만 지원하지만, 경량성과 높은 시장 점유율(2025년 기준 66%)을 바탕으로 다양한 개발자 풀이 존재합니다. Spring Boot 기반 신규 개발의 경우, Tomcat, Undertow, Jetty 등 내장 서버를 활용하여 개발과 배포의 효율성을 극대화할 수 있습니다.

둘째, EJB, JMS, JCA 등 Java EE Full Profile 기능을 사용하는 엔터프라이즈 애플리케이션은 WildFly(구 JBoss) 또는 Open Liberty가 적합합니다. WildFly는 Jakarta EE 10 Full Profile을 지원하며, EJB, JMS, JCA, JPA 등 엔터프라이즈 기능을 모두 제공하고 있습니다. Open Liberty 역시 Jakarta EE 10을 지원하며, 경량화와 클라우드 네이티브 환경에 최적화된 구조를 갖추고 있습니다.

셋째, EJB를 유지하면서 오픈소스 전환을 고려하는 경우에는 WildFly와 Red Hat JBoss EAP(상용 지원)를 조합하는 것이 실무적으로 안정적입니다. JBoss EAP는 WildFly 기반의 상용 제품으로, 엔터프라이즈 환경에서 기술 지원과 SLA를 보장받을 수 있습니다.

마지막으로, Spring Boot 기반 신규 개발은 내장 서버(Tomcat, Undertow, Jetty)를 활용하여 클라우드 네이티브, 마이크로서비스 아키텍처(MSA)에 최적화할 수 있습니다. 이 방식은

CI/CD, 컨테이너화, Kubernetes 배포 등 현대적 개발 환경과 자연스럽게 통합됩니다.

실무에서는 애플리케이션의 기능 요구사항, 개발자 풀이 갖춘 기술 스택, 운영팀의 역량, 상용 지원 필요 여부를 종합적으로 고려하여 WAS를 선택해야 합니다. 예를 들어, 단순한 웹 서비스는 Tomcat으로, 복잡한 엔터프라이즈 서비스는 WildFly 또는 Open Liberty로, EJB를 반드시 유지해야 하는 환경은 JBoss EAP로 전환하는 것이 바람직합니다.

또한, 최근에는 클라우드 네이티브 환경에 적합한 WAS 선택이 중요해지고 있습니다. 예를 들어, 컨테이너 기반 배포와 마이크로서비스 아키텍처를 도입하려는 조직에서는 경량화와 빠른 기동, 유연한 확장성을 제공하는 WAS가 선호됩니다. Tomcat과 같은 경량 WAS는 빠른 배포와 스케일링에 유리하며, Spring Boot 내장 서버는 DevOps 및 CI/CD 파이프라인과의 연계가 용이합니다. 반면, 엔터프라이즈 기능이 필수적인 대규모 시스템에서는 WildFly, Open Liberty와 같이 Jakarta EE Full Profile을 지원하는 WAS가 요구됩니다. 이들 WAS는 트랜잭션 관리, 보안, 메시징, 분산 환경 지원 등 복잡한 엔터프라이즈 요구사항을 충족할 수 있습니다.

실제 사례로, 금융권에서는 대규모 트랜잭션 처리와 높은 가용성이 요구되어 WildFly와 JBoss EAP를 도입하는 경우가 많으며, 스타트업이나 인터넷 서비스 기업에서는 빠른 개발과 배포를 위해 Tomcat 또는 Spring Boot 내장 서버를 선호합니다. 이처럼 조직의 IT 전략, 인력 구성, 예산, 기술 지원 요구 등을 종합적으로 고려하여 WAS를 선정하는 것이 성공적인 오픈소스 전환의 핵심입니다.

5.1.2 상용 기술 지원과 SLA 요구 사항

오픈소스 WAS는 기본적으로 무료이지만, 엔터프라이즈 환경에서는 상용 기술 지원과 SLA가 중요한 고려 요소입니다. WebLogic과 같은 상용 WAS에서 오픈소스 WAS로 전환할 때, 기술 지원 범위와 비용, SLA 수준을 비교 분석해야 합니다.

Red Hat JBoss EAP는 WildFly 오픈소스 프로젝트를 기반으로 한 상용 제품으로, 엔터프라이즈급 기술 지원과 SLA를 제공합니다. Red Hat은 24x7 지원, 긴급 패치, 보안 업데이트, 인증된 운영체제 및 JDK 매트릭스, 장기 지원(LTS) 버전 등을 제공하며, 연간 라이선스 비용은 WebLogic 대비 약 30~50% 수준으로 TCO(Total Cost of Ownership)를 크게 절감할 수 있습니다.

IBM WebSphere Liberty 역시 Open Liberty 기반 상용 제품으로, IBM의 글로벌 기술 지원과 SLA를 제공합니다. WebSphere Liberty는 클라우드 네이티브 환경에 최적화되어 있으며, Jakarta EE 10 지원, 고가용성, 성능 튜닝, 보안 패치 등을 보장합니다. 비용은 Red Hat JBoss

EAP와 유사한 수준이지만, IBM의 글로벌 지원망과 연계된 서비스가 강점입니다.

반면, Apache Tomcat, WildFly, Open Liberty 등 오픈소스 WAS는 커뮤니티 지원이 기본이며, 상용 기술 지원이 필요할 경우 Tomitribe(For TomEE), Red Hat(For WildFly/JBoss), IBM(For Liberty) 등과 별도 계약을 체결해야 합니다. SLA 수준은 계약 조건에 따라 다르며, 보안 패치, 긴급 장애 대응, 운영체제/JDK 인증 매트릭스, 장기 지원 버전 제공 등이 주요 항목입니다.

Oracle WebLogic 라이선스와 비교하면, 오픈소스 WAS로 전환 시 라이선스 비용이 대폭 절감되고, 기술 지원 옵션을 선택적으로 활용할 수 있어 TCO 측면에서 유리합니다. 그러나 자체 기술 역량이 부족한 경우에는 상용 지원 계약을 통해 안정적인 운영을 보장받는 것이 바람직합니다. 실무에서는 비용, 지원 범위, SLA 수준, 벤더 종속성, 기술 유연성을 종합적으로 평가하여 최적의 지원 옵션을 선택해야 합니다.

상용 기술 지원 계약을 체결할 경우, 장애 발생 시 신속한 대응과 문제 해결, 보안 취약점 패치, 장기 지원 버전 제공 등 안정적인 운영 환경을 확보할 수 있습니다. 예를 들어, 금융, 공공, 제조 등 미션 크리티컬 시스템에서는 24x7 지원, 4시간 이내 응답, 긴급 패치 제공 등 엄격한 SLA가 요구됩니다. 반면, 자체적으로 운영 역량이 충분한 조직은 커뮤니티 지원만으로도 운영이 가능할 수 있습니다. 최근에는 클라우드 서비스 사업자(CSP)와 연계된 WAS 지원 옵션도 등장하고 있어, 다양한 지원 모델을 비교 분석하는 것이 필요합니다. 실무적으로는 기술 지원 범위(장애 대응, 패치, 업그레이드, 컨설팅 등), 지원 채널(전화, 이메일, 포털), 응답/복구 시간, 비용 구조, 계약 기간 등을 상세히 검토하여 조직에 최적화된 지원 모델을 선택해야 합니다.

5.2 단계별 전환 절차

오픈소스 WAS로의 마이그레이션은 체계적인 단계별 절차를 통해 진행되어야 성공적으로 완수할 수 있습니다. 단순히 WAS만 교체하는 것이 아니라, 애플리케이션 코드, 설정 파일, 배포 방식, 운영 환경 등 전반에 걸쳐 변화가 수반되므로, 각 단계별로 명확한 목표와 검증 방법을 마련하는 것이 중요합니다. 전환 절차는 일반적으로 1단계: 현행 시스템 분석 및 전환 범위 확정, 2단계: WebLogic 전용 코드 제거 및 표준 API 전환, 3단계: 대상 WAS 환경 구성 및 배포, 4단계: 검증, 성능 테스트, 병행 운영의 4단계로 구성됩니다. 각 단계에서는 자동화 도구 활용, 난이도별 애플리케이션 분류, CI/CD 파이프라인 구축, 병행 운영 전략 등 실무적 방법론이 적용되어야 하며,

전환 과정에서 발생할 수 있는 리스크를 사전 예측하고 대응 방안을 마련하는 것이 필수적입니다.

5.2.1 1단계: 현행 시스템 분석 및 전환 범위 확정

마이그레이션의 첫 단계는 현행 시스템의 구조와 기능을 정확히 분석하고, 전환 범위를 확정하는 것입니다. WebLogic 환경에서 운영 중인 애플리케이션의 기술 스택, 사용 중인 WebLogic 전용 API, 배포 구조, 연동 시스템 등을 면밀히 파악해야 합니다.

정적 분석 도구(RHAMT, MTA, Windup 등)를 활용하여 소스 코드와 설정 파일을 자동 분석할 수 있습니다. 이 도구들은 WebLogic 전용 API 사용량, 배포 기술자(weblogic.xml, weblogic-ejb-jar.xml 등) 의존성, JNDI Lookup, JMS, JDBC, EJB 사용 여부 등을 리포트로 생성하여, 전환 난이도를 객관적으로 평가할 수 있습니다.

분석 결과를 바탕으로 애플리케이션을 난이도별로 분류(Green/Yellow/Red)하는 것이 실무적으로 효과적입니다. Green은 표준 API만 사용하는 전환 난이도 낮은 애플리케이션, Yellow는 일부 WebLogic 전용 기능을 사용하는 중간 난이도, Red는 EJB, JMS, JCA 등 엔터프라이즈 기능과 WebLogic 전용 API를 광범위하게 사용하는 고난이도 애플리케이션입니다.

이 분류를 통해 파일럿 프로젝트 대상 선정, 우선순위 설정, 리소스 배분, 전환 일정 계획 등을 수립할 수 있습니다. 또한, 연동 시스템(DB, MQ, 외부 API 등)과의 호환성, 성능 요구사항, 가용성 요구사항 등 비기능적 요구사항도 함께 정의해야 합니다.

실무에서는 전환팀을 구성하여 개발자, 아키텍트, 인프라 담당자 등 관련 인력이 협업하며, 분석 결과를 바탕으로 상세한 전환 로드맵을 작성해야 합니다. 이 단계에서 전환 범위와 목표 시스템 정의(To-Be 정의), 예산 계획, 리스크 관리 전략을 명확히 수립하는 것이 성공적인 마이그레이션의 출발점입니다.

현행 시스템 분석 단계에서는 기존 시스템의 아키텍처 다이어그램 작성, 주요 컴포넌트 및 인터페이스 목록화, 데이터 흐름 및 트랜잭션 경로 파악, 외부 연동 시스템(예: 인증, 결제, 메시징, 파일 연동 등)과의 연결 구조 분석이 필수적입니다. 또한, 운영 중인 서버의 OS, JDK, 미들웨어, 데이터베이스, 네트워크 환경 등 인프라 현황도 함께 조사해야 합니다. 실무적으로는 시스템 운영 문서, 배포 스크립트, 장애 이력, 성능 모니터링 데이터 등 다양한 자료를 활용하여 현황을 입체적으로 파악해야 하며, 이를 바탕으로 전환 범위(전체/부분, 단계별/일괄 등)와 우선순위, 예상 리스크(예: 레거시 코드, 미지원 API, 성능 병목 등)를 도출할 수 있습니다. 이 단계에서 충분한

분석과 명확한 범위 정의가 이루어져야 이후 전환 작업의 효율성과 성공률이 크게 향상됩니다.

5.2.2 2단계: WebLogic 전용 코드 제거 및 표준 API 전환

두 번째 단계는 WebLogic 전용 코드를 제거하고, Java EE/Jakarta EE 표준 또는 Spring 프레임워크 코드로 전환하는 작업입니다. WebLogic 환경에서 사용되는 독점 API, 배포 기술자, 설정 파일, Annotation 등을 표준 방식으로 이관해야 합니다.

대표적으로 weblogic.xml과 같은 WebLogic 전용 배포 기술자는 web.xml 표준 설정으로 이관해야 합니다. WebLogic JNDI Lookup 코드는 Spring DI(@Autowired) 또는 CDI(@Inject)로 전환하여, 의존성 주입 방식의 현대적 구조로 개선할 수 있습니다.

WebLogic ApplicationLifecycleListener와 같은 전용 라이프사이클 관리 기능은 @Startup, @PostConstruct, @PreDestroy 등 표준 Jakarta EE Annotation으로 대체할 수 있습니다. WebLogic 전용 Annotation(예: @WeblogicEJB, @WeblogicResource 등)은 Jakarta EE 표준 Annotation(@Stateless, @Resource 등)으로 전환해야 하며, 설정 파일 역시 표준 형식으로 변환해야 합니다.

EJB, JMS, JCA 등 엔터프라이즈 기능을 사용하는 경우에는 WildFly, Open Liberty 등 Full Profile WAS로의 전환이 필요하며, EJB 코드의 경우 Spring @Service, CDI @ApplicationScoped 등으로 리팩토링할 수 있습니다. JNDI Lookup 하드코딩, WebLogic 전용 배포 기술자, Work Manager, Coherence 등 독점 기능은 대체 구현 또는 제거가 필요합니다.

실무에서는 코드 리팩토링, 설정 파일 변환, 테스트 코드 작성, 표준 API 적용 등 작업을 병행하며, 자동화 도구와 수동 검토를 통해 호환성 문제를 최소화해야 합니다. 이 단계에서 발생하는 주요 이슈는 클래스로더 충돌, 라이브러리 버전 불일치, 트랜잭션 관리 방식 변경 등이며, 각 이슈에 대해 대안과 검증 방법을 마련해야 합니다.

실제 전환 사례를 살펴보면, WebLogic 전용 JMS 리소스 설정을 WildFly의 standalone.xml 또는 Open Liberty의 server.xml로 변환하는 작업, WebLogic의 Work Manager를 표준 Java EE Concurrency Utilities로 대체하는 작업 등이 포함됩니다. 또한, WebLogic에서 제공하는 Coherence 캐시 기능을 Redis, Hazelcast 등 오픈소스 캐시 솔루션으로 대체하는 경우도 많습니다. 코드 전환 시에는 기존 비즈니스 로직의 변경을 최소화하면서, 표준 API로의 호환성을 확보하는 것이 중요합니다. 테스트 자동화 도구(JUnit, Mockito 등)를 활용하여 리팩토링 후에도

기존 기능이 정상 동작하는지 검증해야 하며, 코드 리뷰와 병행하여 품질을 확보해야 합니다. 이 단계에서 발생하는 주요 리스크는 미지원 기능, 설정 누락, 성능 저하 등이 있으므로, 각 항목별 체크리스트를 마련하여 체계적으로 대응하는 것이 바람직합니다.

5.2.3 3단계: 대상 WAS 환경 구성 및 배포

세 번째 단계는 전환 대상 WAS 환경을 구성하고, 애플리케이션을 배포하는 작업입니다. 선택한 WAS(WildFly, Tomcat, Open Liberty 등)에 맞게 도메인/서버 설정, 데이터소스, JMS, 보안 구성을 수행해야 합니다.

실무에서는 CI/CD 파이프라인(Jenkins, GitLab CI 등)을 구축하여, 소스 코드 관리(Git), 빌드(Maven/Gradle), 테스트, Docker 이미지 빌드, Registry Push, Kubernetes 배포까지 자동화된 배포 환경을 마련할 수 있습니다. 이 과정에서 정적 분석(SonarQube), 테스트 자동화, 배포 자동화, 모니터링 도구(Prometheus, Grafana, ELK Stack 등) 연동을 통해 운영 효율성과 품질을 높일 수 있습니다.

데이터소스 설정은 표준 JDBC DataSource로 이관하고, JMS 설정은 대상 WAS의 표준 방식으로 재구성해야 합니다. 보안 영역은 JAAS, Spring Security, Jakarta EE Security 등 표준 방식으로 전환하며, WebLogic 전용 보안 Realm/Provider는 대체 구현 또는 제거가 필요합니다.

클러스터링, 로드 밸런싱, 고가용성(HA) 구성 등 엔터프라이즈 환경에서는 대상 WAS의 기능을 활용하여 안정성을 확보해야 하며, Kubernetes 기반 배포에서는 Pod, Service, Ingress, ConfigMap, Secret 등 리소스 정의와 함께 스케일링, 롤링 업데이트, 장애 복구 전략을 마련해야 합니다.

실무에서는 배포 자동화, 운영 모니터링, 장애 대응, 백업/복구 전략을 함께 수립하여, 전환 후 안정적인 운영을 보장해야 합니다.

구체적으로, Tomcat이나 WildFly 환경에서는 데이터소스 설정을 context.xml 또는 standalone.xml에 표준 방식으로 등록하고, 보안 설정은 LDAP, OAuth2, SAML 등 외부 인증 시스템과 연동하여 구현할 수 있습니다. 또한, CI/CD 파이프라인 내에서 환경별(개발, 테스트, 운영) 설정 분리, 시크릿 관리, 롤링 배포, 헬스 체크, 자동 롤백 등 고도화된 배포 전략을 적용할 수 있습니다. Kubernetes 환경에서는 Helm Chart를 활용한 배포 자동화, Prometheus와 Grafana를 통한 실시간 모니터링, ELK Stack을 활용한 로그 집계 및 분석, 장애 발생 시 자동 복구(셀프힐링)

등 클라우드 네이티브 운영 체계를 구현할 수 있습니다. 실무에서는 배포 후 운영팀과 개발팀이 협업하여 장애 대응 시나리오, 백업 및 복구 정책, 보안 정책, 운영 매뉴얼 등을 함께 마련하는 것이 중요합니다.

5.2.4 4단계: 검증, 성능 테스트, 병행 운영

마지막 단계는 전환된 시스템의 기능 검증, 성능 테스트, 보안 검증, 병행 운영 전략을 수행하는 것입니다. 기능 검증은 회귀 테스트를 통해 기존 시스템과 동일한 기능이 정상 동작하는지 확인하며, 성능 검증은 부하 테스트를 통해 병목 현상, 응답 시간, 처리량 등을 측정하여 기준선을 재설정합니다.

보안 검증은 취약점 스캔, 침투 테스트, 보안 패치 적용 등을 통해 시스템의 보안성을 확인하며, 운영팀과 개발팀이 협업하여 장애 복구 테스트, 사용자 인수 테스트(UAT) 등을 병행합니다.

병행 운영은 Blue-Green 또는 Canary 배포 전략을 활용하여, 구 시스템과 신 시스템을 동시에 운영하면서 안정적으로 전환을 완료할 수 있습니다. Blue-Green 배포는 두 환경을 병렬로 운영하며 트래픽을 점진적으로 전환하는 방식이고, Canary 배포는 일부 사용자에게만 신 시스템을 먼저 적용하여 점진적으로 확장하는 방식입니다.

실무에서는 모니터링 강화, 문제 해결 및 최적화, 문서화 등 후속 작업을 병행하며, 전환 과정에서 발생하는 이슈를 신속하게 해결하고 시스템을 최적화하는 데 집중해야 합니다. 전환 후에는 지속적인 유지보수와 업데이트, 기술 지원 전략, 인력 교육 및 기술 내재화 등을 통해 안정적인 운영과 미래 지향적인 IT 환경을 구축할 수 있습니다.

이 단계에서는 JMeter, Gatling, LoadRunner 등 부하 테스트 도구를 활용하여 실제 트래픽 시나리오에 기반한 성능 검증을 실시하고, 취약점 진단 도구(OpenVAS, Nessus 등)로 보안 검증을 수행합니다. 장애 복구 테스트에서는 장애 상황(서버 다운, 네트워크 단절 등)을 인위적으로 발생시켜 자동 복구 및 알림 체계를 점검합니다. 병행 운영 중에는 실시간 모니터링을 통해 신·구 시스템의 성능, 오류, 사용자 경험을 비교 분석하고, 문제가 발생할 경우 신속하게 롤백할 수 있는 절차를 마련해야 합니다. 또한, 전환 완료 후에는 운영 매뉴얼, 장애 대응 매뉴얼, 시스템 아키텍처 문서 등 각종 문서화를 철저히 하여, 향후 유지보수와 인수인계가 원활하게 이루어지도록 해야 합니다. 실무적으로는 전환 후 일정 기간 집중 모니터링, 운영팀/개발팀 합동 점검, 사용자 피드백 수집 및 반영 등 후속 관리 체계를 구축하는 것이 안정적인 전환의 핵심입니다.

5.3 개발 및 배포 환경 현대화

오픈소스 WAS로의 전환은 단순히 WAS 교체에 그치지 않고, 개발 및 배포 환경의 현대화까지 포함해야 진정한 기술 부채 해소와 운영 효율성을 달성할 수 있습니다. 기존 WebLogic 환경에서 흔히 사용되던 FTP 기반 수동 배포 방식은 버전 관리 부재, 롤백 불가, 감사 추적 불가 등 심각한 운영 리스크를 내포하고 있습니다. 이에 따라, Git 기반 형상관리와 CI/CD 자동 배포 파이프라인, 컨테이너화, Kubernetes 배포 등 현대적 개발·운영 체계로의 전환이 필수적입니다. 본 절에서는 FTP 배포에서 CI/CD 파이프라인으로의 전환, 컨테이너화와 Kubernetes 배포 방법론을 구체적으로 설명합니다.

5.3.1 FTP 배포에서 CI/CD 파이프라인으로의 전환

기존 WebLogic 환경에서 FTP 기반 수동 배포는 버전 관리가 불가능하고, 롤백이나 감사 추적이 어려워 운영 리스크가 매우 높습니다. 수동 배포는 개발자가 직접 WAR/EAR 파일을 FTP로 전송하고, 운영팀이 수동으로 배포하는 방식이 일반적이지만, 이 과정에서 버전 불일치, 배포 오류, 롤백 실패, 보안 취약점 노출 등 다양한 문제가 발생할 수 있습니다.

현대적 개발 환경에서는 Git 기반 형상관리 체계를 도입하여, 소스 코드의 버전 관리, 브랜치 전략, 협업 개발, 코드 리뷰, 자동 빌드/테스트/배포를 일관되게 수행할 수 있습니다. Jenkins, GitLab CI 등 CI/CD 도구를 활용하여, 소스 코드 커밋 → 빌드(Maven/Gradle) → 테스트 → 배포까지 자동화된 파이프라인을 구축할 수 있습니다.

브랜치 전략(Git Flow, GitHub Flow 등)을 적용하여, 개발/운영 분리, 릴리즈 관리, 긴급 수정, 롤백 등을 체계적으로 관리할 수 있으며, 배포 자동화는 운영팀의 부담을 줄이고, 배포 오류를 최소화할 수 있습니다. 또한, 감사 추적이 가능하여 보안 및 컴플라이언스 요구사항을 충족할 수 있습니다.

실무에서는 CI/CD 파이프라인 구축을 통해 개발 생산성, 운영 효율성, 품질 관리, 보안성을 동시에 달성할 수 있으며, 오픈소스 WAS 환경에서 Jenkins, GitLab CI, GitHub Actions 등 다양한 도구와 연동하여 배포 자동화를 구현할 수 있습니다.

구체적으로, CI/CD 파이프라인은 소스 코드 커밋 시 자동으로 빌드, 테스트, 정적 분석, 패키

징, 배포까지 일련의 과정을 자동화합니다. 예를 들어, 개발자가 Git 저장소에 코드를 푸시하면 Jenkins가 자동으로 빌드와 테스트를 수행하고, 성공 시 WAR/EAR 파일을 생성하여 운영 환경에 자동 배포합니다. 이 과정에서 SonarQube를 통한 코드 품질 분석, Slack/Teams를 통한 알림, Jira와의 연동을 통한 이슈 추적 등 다양한 자동화가 가능합니다. 또한, 롤백 기능을 통해 배포 실패 시 이전 버전으로 신속하게 복구할 수 있으며, 배포 이력과 변경 내역이 모두 기록되어 추적성과 감사성이 확보됩니다. 실무에서는 CI/CD 파이프라인을 통해 배포 주기를 단축하고, 휴먼 에러를 최소화하며, DevOps 문화 정착과 품질 향상에 크게 기여할 수 있습니다.

5.3.2 컨테이너화와 Kubernetes 배포

오픈소스 WAS 기반 애플리케이션을 Docker 이미지로 패키징하고 Kubernetes에 배포하는 것은 현대적 클라우드 네이티브 환경의 핵심입니다. 기존 WebLogic의 도메인 기반 클러스터링은 서버 인스턴스, 도메인, 클러스터, 데이터소스, JMS 등 복잡한 설정이 필요하지만, Kubernetes의 Pod 기반 스케일링은 컨테이너 단위로 배포와 확장이 가능하여 운영 효율성이 크게 향상됩니다.

Git 기반 형상관리 → Build(Maven/Gradle) → 정적 분석(SonarQube) → 테스트 → Docker Image Build → Registry Push → Kubernetes 배포 파이프라인을 구축하면, 개발자가 소스 코드를 커밋하면 자동으로 빌드, 테스트, 이미지 생성, 배포까지 일관된 프로세스가 실행됩니다. 이 과정에서 버전 관리, 롤백, 감사 추적, 보안 검증이 자동화되어 운영 리스크를 최소화할 수 있습니다.

Kubernetes 배포에서는 Pod, Service, Ingress, ConfigMap, Secret 등 리소스를 정의하여, 스케일링, 롤링 업데이트, 장애 복구, 모니터링 등 현대적 운영 체계를 구현할 수 있습니다. WebLogic의 도메인 기반 클러스터링과 비교하면, Kubernetes는 Pod 단위로 스케일링이 가능하고, 서비스 디스커버리, 로드 밸런싱, 자동 복구 등 클라우드 네이티브 환경에 최적화되어 있습니다.

컨테이너화는 향후 클라우드 전환의 기반이 되며, 마이크로서비스 아키텍처, DevOps, CI/CD, IaC(Infrastructure as Code) 등 현대적 IT 트렌드와 자연스럽게 연계됩니다. 실무에서는 Docker, Kubernetes, Helm, Prometheus, Grafana, ELK Stack 등 오픈소스 도구를 활용하여, 개발·운영 환경을 현대화하고, 기술 부채를 해소할 수 있습니다.

실제 사례로, 대형 이커머스 기업에서는 WAS 기반 애플리케이션을 Docker 이미지로 패키징한 후, Kubernetes 클러스터에 배포하여 수십~수백 개의 인스턴스를 자동으로 확장·축소하고, 롤링 업데이트와 블루-그린 배포 전략을 적용하여 무중단 배포를 실현하고 있습니다. 또한, ConfigMap

과 Secret을 활용하여 환경별 설정과 민감 정보를 안전하게 관리하고, Prometheus와 Grafana로 실시간 모니터링, ELK Stack으로 로그 집계 및 분석을 수행합니다. 이러한 환경에서는 장애 발생 시 자동 복구(셀프힐링), 오토스케일링, 서비스 디스커버리 등 클라우드 네이티브의 이점을 극대화할 수 있습니다. 실무적으로는 Helm Chart를 활용한 배포 템플릿 관리, GitOps(ArgoCD, Flux 등) 기반 선언적 배포, IaC(Terraform, Ansible 등)를 통한 인프라 자동화 등 고도화된 운영 체계를 도입하여, 개발·운영 효율성과 확장성을 동시에 확보할 수 있습니다.

참고자료

- Apache Tomcat 공식 문서
- 비싼 WAS의 시대는 끝났다! 클라우드 네이티브 최적화 WAS로 전환할 때
- 오픈마루 JBoss 운영 노하우
- JBoss, Tomcat, JEUS, Weblogic, WebSphere 비교

Contact Us

hello@cncf.co.kr

02-469-5426

www.cncf.co.kr

CNF Blog

다양한 콘텐츠와 전문 지식을 통해
더 나은 경험을 제공합니다.

CNF eBook

이제 나도 클라우드 네이티브 전문가
쿠버네티스 구축부터 운영 완전 정복

CNF Resource

Community Solution의 최신 정보와
유용한 자료를 만나보세요.

씨엔에프 | CNF

전화 : (02)469-5426
팩스 : (02)469-7247
메일 : hello@cncf.co.kr