ChatGPT

현실적인 소스 관리 및 배포 전략 제안

그림: Git Flow 브랜치 모델 예시. `master`는 운영(프로덕션) 릴리스를, `develop`는 차기 버전 개발을, `release` 브랜치는 배포 준비를, `feature` 브랜치는 개별 기능 개발을, `hotfix` 브랜치는 긴급 수정을 담당한다.

1. 브랜치 전략

현재 **feature 브랜치 -> develop 브랜치**로 모두 병합한 뒤 배포하고 있어, 개발 완료되었지만 **운영에 내보내지 않을 기능도 develop에 포함**되는 문제가 있습니다. 이를 해결하려면 **운영(Main)용 브랜치**와 **개발(Develop)용 브랜치**를 분리하는 **브랜치 전략**이 필요합니다. 일반적으로 많이 사용하는 Git Flow 모델을 응용하는 것을 권장합니다 [1]. 주요 브랜치 전략은 다음과 같습니다:

- **메인 브랜치(`master` or `main`)** – **운영 배포용 안정 브랜치**로서, 운영 서버에 반영된 **검증된 코드만 유지**합니다 [2]. 이 브랜치의 커밋들은 버전 태그를 붙여 관리하면 좋습니다 (예: `v1.0.0`) [2].
- **개발 브랜치(`develop`)** – **통합 개발 브랜치**로서, 여러 기능을 병합하여 **다음 출시 버전**을 준비합니다 [2] [3]. 개발 완료된 기능(feature 브랜치)을 코드 리뷰 및 기본 테스트 후 이 브랜치에 병합하고, 이 브랜치 기반으로 테스트 서버에 배포해 통합 테스트를 진행합니다. 단, **운영에 포함되지 않을 기능은 이 브랜치에 병합을 보류**하는 것이 핵심입니다 [4] [5]. 즉, 현재 릴리스에 넣지 않을 기능이라면 develop에 아직 합치지 않고 다음 릴리스까지 **feature 브랜치 상태로 유지**합니다. 이를 위해 코드 병합은 Pull Request 등을 통해 **승인된 것만 진행**하고, 필요시 develop 브랜치를 일시적으로 **동결(lock)**하여 릴리스 직전 불필요한 병합을 막을 수 있습니다 [4].
- **기능 브랜치(`feature/*`)** – 각 기능이나 이슈별로 분기하여 작업하는 브랜치입니다. **항상 `develop` 브랜치로부터 분기**하며, 기능 개발 완료 후 PR 등의 절차를 거쳐 develop에 병합됩니다 [6] [7]. 운영에 포함되지 않기로 한 기능이라면 feature 브랜치를 계속 유지하고 develop에 합치지 않으며, 나중에 필요해질 때 병합하거나 폐기합니다 [5].
- **릴리즈 브랜치(`release/*`)** – 운영 배포를 준비하기 위한 **임시 브랜치**로, 이번 배포에 포함될 기능만 담은 코드를 안정화합니다. 보통 **develop에서 분기**하며, 이 시점부터는 **새 기능 추가는 중단**하고 테스트/버그 수정을 진행합니다 [8]. 배포 준비가 끝나면 해당 release 브랜치를 **`master`에 병합하고 태그를 달아** 릴리스합니다 [8] [9]. 또한 릴리즈 과정에서 수정된 내용이 있다면 release 브랜치를 다시 develop에도 병합하여 이후 이력이 일치하도록 합니다 [10]. 릴리즈 브랜치는 배포 완료 후 삭제합니다. 이 방식으로 **필요한 기능만 추려서 배포**할 수 있고, 개발팀은 병행하여 develop에서 다음 기능 작업을 계속할 수 있습니다 [11] [12].
- **핫픽스 브랜치(`hotfix/*`)** – 운영 중 발생한 긴급 버그를 수정하기 위한 브랜치로, **`master`에서 바로 분기**합니다. 빠른 패치를 위해 사용하며, 완료 시 `master`와 `develop`에 모두 병합합니다 [13] [14]. 이 브랜치를 통해 운영 이슈를 급히 해결하면서도 develop의 코드와 **차이를 최소화**할 수 있습니다.

위와 같은 전략(일종의 **Git Flow** 모델)을 적용하면, **테스트용 develop 브랜치**와 **운영 배포용 master 브랜치** 간에 배포 대상을 명확히 구분할 수 있습니다 [1]. 핵심은 **운영에 내보낼 준비가 된 코드만 master에 존재**하게 하고, 그렇지 않은 코드는 develop (또는 feature 브랜치)에 머물러 있도록 하는 것입니다. 이로써 **운영 배포 시 의도치 않은 기능이 섞여 들어가는 일을 원천 방지**합니다.

2. 운영/테스트 배포 관리 방식

테스트 환경과 운영 환경의 반영 범위를 명확히 분리하여 관리해야 합니다. 현재는 테스트 서버(동일 서버의 테스트 인스턴스)에 모든 기능이 배포되기 때문에, 운영에는 그 중 일부만 내보내려 해도 war 파일에 불필요한 코드가 남아 문제를 일으키고 있습니다. 이를 개선하는 배포 관리 방안은 다음과 같습니다:

- **브랜치별 빌드/배포 분리:** 앞서 정한 브랜치 전략에 따라, **테스트 서버에는** `develop` **브랜치 빌드(war)**를 배포하고 **운영 서버에는** `master` **(혹은 준비된 release 브랜치) 빌드**를 배포합니다. 예를 들어, develop 브랜치에서 최신 코드를 빌드해 **test.war**로 배포하고 통합 테스트를 진행합니다. 운영 배포 시에는 master 브랜치 기준으로 **prod.war**를 생성하여 배포합니다. 이렇게 하면 **운영용 war에는 승인된 기능만 포함**되고, test.war에는 향후 배포될 모든 기능이 포함될 수 있습니다.
- **동일 서버에서 환경 격리:** 물리적으로 같은 서버를 사용해야 한다면, **Tomcat 인스턴스를 분리**하거나 **Context를 분리**하여 하나는 테스트용, 다른 하나는 운영용으로 사용합니다. 예를 들어, 하나의 Tomcat에 두 개의 웹 애플리케이션 컨텍스트(예: `/app_test` vs `/app`)를 두거나, Tomcat을 두 대 띄워 포트만 다르게 운영할 수 있습니다. 이렇게 하면 **테스트용 WAR과 운영용 WAR**을 동시에 올려둘 수 있고, 경로 또는 포트로 구분하여 접근합니다. 이때 각 환경에 맞는 **DB 연결 정보나 설정**도 별도로 관리해야 하며, 프로퍼티 파일 등을 분리하여 패키징합니다 (Maven Profile 등을 활용 가능).
- **수동 배포 프로세스 표준화:** 수동으로 war를 옮겨 배포하더라도, **절차를 표준화**하여 실수를 줄입니다. 예를 들어, 운영 배포 전에는 항상 **master 브랜치의 특정 태그 커밋을 빌드**하도록 규칙화하고, 빌드 산출물(war)에 **버전명을 포함**시킵니다 (예: `MyApp-1.2.3-prod.war`). 테스트 배포 시에도 마찬가지로 `develop` 최신 커밋이나 특정 릴리즈 후보 커밋을 빌드하여 `MyApp-1.2.3-rc.war` 처럼 이름을 붙입니다. 이렇게 하면 **어느 WAR 파일이 어떤 버전/브랜치의 코드인지 식별**이 쉬워집니다. 또한 운영 배포 시 **이전 버전 war를 백업**해 두고, 새 버전을 배포한 뒤 문제 발생 시 롤백할 수 있는 대비책도 함께 수립합니다.
- **배포 내역 문서화:** 테스트 서버에 어떤 기능들이 반영되었고, 운영에는 그 중 어떤 기능만 배포했는지 **변경 내역을 기록**해두면 좋습니다. 예를 들어, 스프레드시트나 변경 로그를 통해 **테스트 배포본 vs 운영 배포본의 차이**를 명시하거나, Git의 릴리즈 노트를 활용해 해당 배포 태그에 포함된 커밋 목록을 공유합니다. 이렇게 하면 클라이언트에게도 **"테스트 환경에는 A, B, C 기능이 있으나 이번 운영 배포에는 A, B만 포함"**이라고 명확히 설명할 수 있고, 팀 내부적으로도 추후 어느 버전을 배포했는지 혼동을 줄일 수 있습니다.
- **동일 서버 사용 시 주의:** 테스트와 운영이 같은 서버에서 돌아갈 경우, **리소스 경합과 설정 충돌 방지**에 신경써야 합니다. 예를 들어 메모리, CPU 등이 한쪽 배포로 인해 소모되어 다른 쪽에 영향 주지 않도록 하고, 테스트 중인 기능이 운영 데이터베이스를 참조하거나 하지 않도록 **철저히 분리된 설정**을 사용해야 합니다. 가능하다면 운영과 테스트 **데이터베이스도 분리**하는 것이 안전하며, 동일 DB를 써야 한다면 쿼리 등이 서로 간섭하지 않도록 주의합니다.

결론적으로, **테스트 배포는 "차기 버전 전체"를 대상으로, 운영 배포는 "검증 완료된 선택된 기능만"** 대상으로 삼아, **각각 다른 브랜치에서 빌드된 war를 사용**해야 합니다. 이렇게 하면 테스트 환경에서 여러 기능을 합쳐 시험하면서도, 실제 운영 배포 시에는 원하는 기능만 추려 안정적으로 배포할 수 있습니다.

3. 무료 솔루션을 활용한 구성 방법

예산이나 인프라 제약으로 **자동화 도입이 어려운 상황**이라 해도, 몇 가지 **무료 도구와 방법**을 활용하여 배포 실수를 줄이고 효율을 높일 수 있습니다:

- **GitHub + Git** : 현재 GitHub를 사용하고 있으므로, **GitHub의 무료 기능**들을 최대한 활용합니다. 예를 들어 **Git 브랜치 보호(Branch Protection)** 설정을 통해 `master`나 `develop` 브랜치에 **임의 커밋/푸시를 금지**하고, **Pull Request 병합만 허용**하도록 할 수 있습니다. 이는 실수로 잘못된 코드를 직접 push하거나, 승인되지 않은 코드를 병합하는 일을 방지해 줍니다. 또한 **Git 태그**와 **GitHub Releases** 기능도 무료로 사용할 수 있으므로, 운영 배포 시 해당 커밋에 버전 태그를 달고(GitHub Release를 만들어 배포 노트를 남기는 등) 기록해두면 유용합니다.

- **Jenkins (오픈소스 CI 서버)** : Jenkins는 **오픈소스 자동화 서버**로, 빌드/테스트/배포 과정을 자동화할 수 있습니다 [15] . 무료로 설치하여 사용할 수 있으므로, 사내 허용 범위에서 **빌드 자동화** 정도는 도입할 만합니다. 예를 들어 **Jenkins로 Maven 빌드 작업**을 만들어 두면, 개발자가 수동으로 STS나 IDE에서 패키징하지 않고도 **정해진 브랜치의 최신 코드를 자동으로 빌드하여 WAR 파일을 산출**할 수 있습니다. Jenkins를 사내 개발 PC나 별도 서버에 설치하고 GitHub 저장소와 연동하면, **브랜치별로 CI 파이프라인**을 구축할 수 있습니다 (예: develop 브랜치 푸시 -> 테스트 빌드, master 브랜치 태그 -> 배포용 빌드 등). Jenkins는 결과물을 저장 아티팩트로 보관할 수도 있어, 이전 버전 war를 쉽게 찾아볼 수 있고, 단순 파일 복사 배포의 실수를 줄여줍니다.
- **GitHub Actions** : GitHub에서 제공하는 Actions 역시 **무료로 일정 분량 사용**가능한 CI/CD 도구입니다 [16] . 리포지토리가 private인 경우에도 매월 제한된 런타임 분량 내에서는 무료로 활용할 수 있습니다. GitHub Actions을 활용하면 **클라우드 상에서 자동으로 빌드 및 간단한 테스트**를 수행하고, 결과물 WAR를 artifact로 업로드해 둘 수 있습니다. 예를 들어, 운영 배포용 태그를 만들면 Actions workflow가 돌아 해당 커밋을 빌드하고 WAR를 만들어 줍니다. 비록 **원격 서버에 직접 배포**는 할 수 없는 환경이라 하더라도, **빌드된 WAR를 확실한 절차로 얻을 수 있다는 점**에서 인간 실수를 줄이고 신뢰성을 높입니다. 이렇게 생성된 아티팩트를 사람이 다운로드하여 서버에 올리는 방식으로 **반자동 배포** 흐름을 만드는 것입니다.
- **Maven 및 기타 무료 라이브러리** : Maven 자체도 강력한 무료 빌드 도구이므로, 이를 최대한 활용합니다. `pom.xml`에서 **버전 관리**를 철저히 하여 (`1.2.0-SNAPSHOT` , `1.2.0-RC1` , `1.2.0` 등 단계별 버전) 빌드 결과물에 버전이 반영되게 하고, **Maven Profiles**를 사용하면 **테스트/운영 환경별 설정**(예: DB 연결정보, 특정 기능 on/off)을 프로파일로 나눠 빌드할 수도 있습니다. 또한 **로컬 Nexus 리포지토리(OSS 버전)**나 **JFrog Artifactory OSS** 등을 도입하면, 산출된 WAR를 안전하게 호스팅하며 관리할 수 있습니다 (필수는 아니지만 무료로 구축 가능).
- **Feature Toggle(기능 토글)** : 코드 측면에서는, **특정 기능을 배포하되 운영에서는 비활성화**할 수 있는 **기능 토글 기법**도 무료로 구현 가능합니다. 예를 들어, **Togglz** 같은 오픈소스 자바 라이브러리를 사용하면 새로운 기능마다 토글(스위치)을 붙여 **런타임에 해당 기능을 켜거나 끌 수 있게** 만들 수 있습니다 [17] . 이를 활용하면 코드 자체는 모든 기능이 포함되더라도, 운영 환경에서는 **클라이언트 승인이 된 일부 기능의 토글만 활성화**하고 나머지는 꺼둠으로써, 의도치 않은 기능 동작을 막을 수 있습니다. 이 접근은 **개발/테스트와 운영의 코드 차이를 줄이는 대신, 설정으로 기능을 제어**하는 방식입니다. 다만 토글 사용 시 토글 관리 비용, 복잡도가 증가하므로 팀의 역량과 시간에 따라 도입 여부를 판단해야 합니다. (토글용 설정은 DB나 설정파일로 두고, 운영 배포 시 해당 값만 조절하는 식으로 구현합니다.)

이처럼 **무료 도구들과 Git 자체 기능**만으로도 일정 수준 **프로세스 개선과 실수 예방**이 가능합니다. 초기에는 다소 설정 작업이 필요하지만, 일단 구축해두면 이후부터는 **일관된 방법으로 배포**할 수 있어 사람에 의존한 실수 확률을 크게 줄여줄 것입니다.

4. 유료 솔루션 도입 시 활용 도구 및 이점

만약 **예산 확보나 인프라 개선**이 가능한 상황이라면, 더욱 **견고하고 자동화된 배포 체계**를 구축할 수 있습니다. 몇 가지 도구와 기대 이점을 소개하면 다음과 같습니다:

- **전문 CI/CD 파이프라인 도구**: Jenkins 등으로도 충분하지만, **클라우드 기반 CI/CD 서비스**나 **엔터프라이즈 도구**는 더 편리한 기능과 지원을 제공합니다. 예를 들어 **Azure DevOps (Pipelines)**, **GitLab CI/CD**, **CircleCI**, **TeamCity** 등의 도구는 GUI로 파이프라인을 구성하고 모니터링하기 쉽고, 권한 관리나 비밀 관리 등이 통합되어 있습니다. 이러한 도구를 도입하면 **테스트 -> 스테이징 -> 운영**에 이르는 **다중 단계 배포 파이프라인**을 자동화할 수 있고, **승인 절차**도 설정할 수 있습니다. 특히 Azure DevOps의 Release 파이프라인이나 GitHub Actions의 Environment 기능을 쓰면, 특정 커밋을 **수동 승인 후 운영 서버에 배포**하는 흐름도 구성 가능합니다.
- **배포 자동화/오케스트레이션 도구**: 운영 서버에 직접 SSH가 어렵다면, **에이전트 기반 배포 도구**를 고려할 수 있습니다. **Octopus Deploy** 같은 상용 배포 자동화 도구는 대상 서버에 에이전트를 설치해두고, 중앙에서 배포 명령을 내려 **WAR 파일 배포, 설정 변경, 서비스 재기동** 등을 원클릭으로 수행할 수 있습니다. Octopus Deploy는 **온프레미스 환경에도 배포를 쉽게 자동화**하도록 설계된 도구로, 소프트웨어 배포를 신뢰성 있게 반복 가능하게 만들어줍니다 [18] . 이러한 도구를 도입하면 수동 파일 복사나 원격 데스크톱 접속 없이도 **안전한 배포**가 가능하며, **이력 관리, 롤백**도 체계적으로 지원됩니다.

- **관리형 클라우드 서비스 활용**: 만약 인프라 변경이 허용된다면, **테스트/운영 서버를 클라우드 환경**으로 일부 이전하고 **자동 배포**를 적용할 수도 있습니다. 예를 들어 AWS 환경에서 **CodeDeploy**나 **Beanstalk**, Azure의 **App Service** 등을 쓰면, GitHub와 연동한 푸시 배포나 CI 결과 배포가 가능합니다. 이러한 클라우드 서비스는 자동으로 인스턴스를 관리하고 헬스 체크, 롤백 등을 지원하므로, **배포로 인한 다운타임과 오류를 최소화**할 수 있습니다. 다만 현재 물리 동일 서버를 사용하는 상황이라 당장 큰 변경이 어렵다면, 향후 인프라 투자 시 고려해볼 수 있습니다.
- **분리된 테스트/Staging 환경 구축**: 예산이 있다면 **테스트용 별도 서버**를 마련하여 현재 물리적으로 동일한 서버를 분리하는 것이 이상적입니다. 독립된 테스트(Staging) 서버를 운용하면, 운영 배포 전에 **운영과 동일한 환경에서 최종 리허설**을 해볼 수 있고, 테스트중인 새로운 기능이 운영 환경에 **절대 간섭하지 않도록** 물리적인 격리가 보장됩니다. 이는 비용이 들지만, **운영 안정성** 면에서 얻는 이점이 큽니다.
- **Feature Flag 상용 서비스**: 앞서 언급한 기능 토글을 보다 체계적으로 사용하려면 **LaunchDarkly**, **Split.io**, **Azure App Configuration** 같은 **유료 기능 플래그 서비스**를 도입할 수 있습니다. 이런 서비스들은 **실시간으로 기능 On/Off**를 토글하고, 사용자 그룹별 차등 적용, A/B 테스트 등 고급 기능을 제공합니다. 초기 개발 부담을 줄이고 안전한 릴리스를 가능하게 하지만, 비용이 발생하며 인터넷 연결이 되어야 사용할 수 있다는 점을 고려해야 합니다.
- **기타 협업/품질 도구**: 프로세스 전반을 개선하기 위해 **유료 이슈 트래커(Jira 등)**나 **프로젝트 관리 도구**를 쓰면 배포 대상 기능 선별과 커뮤니케이션이 수월해집니다. 또한 **SonarQube 같은 코드 품질 도구(커뮤니티 에디션은 무료, 유료 버전은 지원 강화)**를 활용하면 배포 전 코드 상태를 점검하여 버그를 사전에 줄일 수 있습니다. 이러한 도구들은 필수는 아니지만, 팀의 규모와 필요에 따라 투자하면 **배포 실패 확률 감소**와 **효율적인 협업**에 기여할 수 있습니다.

유료 솔루션을 도입하면 초기 비용은 들지만, **배포 과정의 신뢰성과 반복 가능성**이 극대화되고 **사람에 의존한 수작업이 크게 줄어드는 효과**가 있습니다. 특히 오류 발생시 **추적과 복구**가 수월하고, 규칙에 따라 선택된 기능만 배포되도록 체계화할 수 있습니다. 만약 회사 정책이 허용하고 예산이 있다면, 장기적 관점에서 이러한 도구 도입을 검토해볼 가치가 충분합니다.

5. Git 기능 활용한 실수 방지 전략 (Cherry-pick, 태그 등)

마지막으로, **Git 자체의 기능**들을 잘 활용하는 것으로도 실수를 예방하고 배포 내역을 명확히 할 수 있습니다. 특히 **cherry-pick과 태그(tag)**는 부분 배포나 이력 관리에 유용하지만, 올바르게 사용하는 것이 중요합니다:

- **Cherry-pick으로 필요한 커밋만 선택 적용:** Cherry-pick은 한 브랜치의 특정 커밋 변경분을 다른 브랜치에 **선택적으로 적용**하는 Git 기능입니다. 예를 들어, develop에 여러 기능이 합쳐졌는데 그 중 한두 커밋만 운영에 반영해야 하는 상황에서, **운영용 브랜치로 해당 커밋들만 cherry-pick**하여 부분 배포를 구현할 수 있습니다 [19]. 그러나 cherry-pick은 **새로운 커밋 이력을 생성**하기 때문에 이후 병합 시 충돌 관리가 어렵고, 이력이 분산되는 단점이 있습니다 [20]. 가능하다면 cherry-pick보다는 앞서 설명한 **릴리즈 브랜치 방식(아예 필요한 것만 있는 브랜치를 생성)**이 낫지만, 부득이 cherry-pick을 쓸 경우 다음 사항을 지킵니다:
- Cherry-pick한 커밋들은 나중에 반드시 원래 개발 브랜치(develop)에 **Merge**하거나 동일한 수정이 적용되도록 해서, **코드베이스 차이**를 최소화합니다. 예를 들어 master에 직접 cherry-pick한 버그수정 커밋은, 이후 develop에도 똑같이 적용하거나 develop -> master 병합 시 충돌을 해결해야 합니다.
- **커밋 단위**로 기능을 잘게 나눠두면 cherry-pick이 수월해집니다. 하나의 기능이 여러 커밋에 걸쳐 섞여 있으면 일부만 선택하기 어렵기 때문에, 평소 PR시 "하나의 기능 = 하나의 커밋"(스쿼시 머지 등) 원칙을 세우면 좋습니다 [21].
- Cherry-pick으로 생성된 커밋들은 원본과 해시가 다르므로, **커밋 메시지에 reference**를 달아 원본을 표시해두면 추적에 도움이 됩니다 (예: "Feat X 적용 (cherry-picked from commit abc1234)").

- **대안:** Cherry-pick 대신, **특정 시점의 커밋을 기준으로 브랜치를 분기**하는 방법도 있습니다. 예컨대 develop에 원치 않는 커밋이 일부 섞였을 경우, 그 이전 커밋 해시를 이용해 release 브랜치를 만들면 해당 부분을 제외하고 배포 준비를 할 수 있습니다 [22]. 이는 여러 커밋을 일일이 cherry-pick하는 것보다 실수를 줄이고 이력을 깨끗이 유지하는 데 유리합니다.

- **Git 태그로 배포 버전 식별:** Git 태그는 특정 커밋을 가리키는 **이름표**로, 배포한 버전을 명확히 표시해줍니다. **운영에 배포할 때마다 태그를 생성**하여 (예: `v1.5.0-prod`, `v1.5.0-rc1`) 해당 커밋을 기록해두세요. Atlassian 등 업계에서도 **공식 릴리스 이력은 main(master) 브랜치 커밋에 태그를 붙여 관리**하는 방식을 권장합니다 [2]. 태그를 활용하면 다음과 같은 이점이 있습니다:

- **배포 이력 관리:** 어떤 코드가 운영에 올라갔는지 한눈에 볼 수 있습니다. GitHub Releases와 연계하면 태그별 변경내역을 자동으로 정리할 수도 있습니다.
- **차이 분석:** 두 태그(배포 버전) 사이의 **Git 비교(diff)**를 통해, 이번 배포에 어떤 변경이 있었는지 쉽게 파악할 수 있습니다. 이는 테스트 환경과 운영 환경의 차이를 검토하거나, 예상치 못한 변경이 포함되지 않았는지 검증하는 데 도움됩니다.
- **롤백 지원:** 만약 새로운 버전에서 문제가 발생하면, 이전 안정 태그로 되돌리는 것이 용이합니다. 태그를 기준으로 코드를 체크아웃해 바로 재배포하거나, 긴급히 hotfix 브랜치를 만들 수도 있습니다.
- **커뮤니케이션:** 태그를 버전 넘버로 관리하면, 개발/QA/운영/클라이언트 간에 **"현재 운영 버전은 v1.4입니다"**처럼 명확한 소통이 가능합니다.

- **주의:** 태그는 기본적으로 한 번 푸시하면 변경하지 않는 것을 원칙으로 합니다(필요 시 동일 이름 태그 강제 갱신은 가능하나 권장되지 않음). 따라서 실수로 잘못된 커밋에 태그를 단 경우 **새 버전 번호로 다시 태그**하고 제대로 배포하는 편이 좋습니다.

- **그 외 Git 활용 팁:**

- **Revert 활용:** 만약 잘못된 기능이 develop에 합쳐졌다면, 이를 제거하는 **revert 커밋**을 만들어 대응할 수 있습니다. 이후 해당 기능이 준비되면 revert를 다시 revert하여 기능을 복구하는 식으로, **브랜치 이력을 관리**할 수 있습니다. 이 방법은 cherry-pick보다 이력 추적이 명확하지만, 잦은 revert는 혼란을 줄 수 있으므로 **최종 릴리스 전에 develop을 정리하는 용도**로 활용합니다 [23].
- **브랜치 보호 및 PR 필수화:** GitHub의 설정으로 특정 브랜치에 **머지하기 전에 PR 리뷰와 승인**을 요구하고, **CI 빌드 통과가 필수**도록 할 수 있습니다. 이를 적용하면 품질이 낮은 코드나 미승인 기능이 실수로 운영 브랜치로 들어가는 것을 막아줍니다.
- **커밋 컨벤션/체크리스트:** 배포 커밋에는 `[Deploy]` 등의 태그를 제목에 붙이거나, 배포 전 체크리스트 (예: 버전 업데이트, 환경 설정 확인 등)를 Git 템플릿/훅으로 관리하면 사람이 잊기 쉬운 부분을 자동으로 상기시킬 수 있습니다.

요약하면, **Git의 선택적 커밋 적용(cherry-pick)**과 **명확한 이력 표시(tag)**를 전략적으로 사용하면 부분 배포와 버전 관리를 안전하게 수행할 수 있습니다. 다만 cherry-pick은 **최후의 수단**으로 생각하고, 가급적 **체계적인 브랜치 전략**으로 필요한 경우를 줄이는 것이 바람직합니다 [24] [25]. 태그는 적극 활용하여 모든 배포를 기록하고, 배포된 소스의 정확한 지점을 식별 가능하게 만들어 두는 것이 좋습니다.

결론: 현실적이고 유지보수하기 쉬운 전략

위에서 논의한 내용을 종합해 보면, 가장 현실적이고 유지보수하기 쉬운 접근은 **Git 브랜치 전략 개선 + 수동 배포 프로세스 체계화**로 요약됩니다. 구체적으로 **Git Flow에 기반한 분기 모델(특히 운영 배포용 브랜치 분리와 release 브랜치 활용)**을 도입하고, 현재의 수동 WAR 배포 방식을 그에 맞춰 조정하는 것을 추천합니다.

이 전략의 절차를 정리하면 다음과 같습니다:

1. **브랜치 운영:** 개발자는 기능별로 **feature 브랜치**를 만들고 작업합니다. 충분히 테스트되고 **운영 배포 승인이 난 기능만 PR을 통해** `develop` **에 병합**합니다. 아직 검증이 덜 되었거나 당장 출시 계획이 없는 기능은 develop에 합치지 않고 대기합니다 [26] [25]. `develop` 브랜치는 항상 **다음 운영 배포候 보낼 후보 기능들로만 구성**되도록 관리합니다.

2. **테스트 배포:** 일정 주기(예: 스프린트 말)나 주요 기능 완료 시점에 **테스트 서버에 배포**합니다. 이때 `develop` 브랜치 최신 커밋으로 WAR를 빌드하여 배포하고, QA와 클라이언트 검수를 진행합니다. 테스트 환경에는 이번 배포 후보 기능들 plus 알파 기능(만약 포함됐더라도)이 있을 수 있으나, **이번 배포에 제외될 기능은 목록을 명시**하여 테스트 시 유의하도록 합니다. 필요하면 **기능 토글**로 제외 기능을 비활성화한 상태로 테스트합니다.
3. **릴리즈 브랜치 생성:** 운영에 특정 버전만 배포하기로 결정되면, `develop`에서 **release 브랜치**를 분기합니다 (예: `release/1.5.0`). 이 브랜치는 오로지 **이번에 운영에 내보낼 기능들만 담긴 상태**여야 합니다 [8]. 만약 develop에 제외해야 할 변화가 섞였으면, **release 브랜치를 분기할 커밋을 develop 이력 중 적절한 지점으로 선택**하거나, release 브랜치에서 해당 커밋을 **revert**하여 조정합니다 [22]. 이렇게 준비된 release 브랜치를 **테스트 서버에 배포 (Staging)**하여 **운영과 동일한 구성으로 최종 검증**을 거칩니다. (현실적으로 별도 Staging 서버가 없다면, 테스트 서버를 이 단계에서 릴리즈 후보 버전으로 업데이트하여 검증할 수도 있습니다.)
4. **운영 배포:** release 브랜치의 검증이 끝나면, 그 코드를 `master` **브랜치에 병합**합니다. 병합한 커밋에는 **태그를 달아 버전 표시**를 합니다 (예: `v1.5.0`). 이제 이 커밋 기준으로 WAR 파일을 빌드하여 **운영 서버에 배포**합니다. 수동으로 할 경우라도 **태그명과 일치하는 커밋인지** 확인하고 진행하며, 이 태그와 배포 일자를 릴리스 노트로 남겨둡니다.
5. **사후 처리:** 운영 배포가 완료되면, release 브랜치를 `develop`에도 병합하여 (테스트 중 발견된 버그 수정 등이 있었다면 develop에 반영) 코드 일치를 유지하고 release 브랜치는 삭제합니다 [11] [27]. 또한 이번 릴리스에서 제외되어 남겨둔 feature 브랜치들은 다음 사이클로 이월하거나, 요구 변경으로 폐기될 경우 정리합니다. 운영에 반영된 커밋들은 모두 태그로 관리되고 있으므로, 이후 문제가 발생하면 해당 버전 태그를 참고해 **신속히 hotfix 브랜치를 만들어 패치**한 뒤 master와 develop에 반영합니다.

위 흐름을 지원하기 위해 **GitHub/CI 도구**를 적절히 활용하면 더욱 좋습니다. 예를 들어 PR 기반 개발 문화, 브랜치 보호, 자동 빌드/테스트 등이 정착되면 자연스럽게 **운영에 넣지 말아야 할 코드가 들어가는 일을 방지**할 수 있습니다. Jenkins나 GitHub Actions로 **브랜치별 WAR 빌드 자동화**를 해두면, 개발자는 해당 WAR를 내려받아 배포만 하면 되므로 편의성과 정확성이 올라갑니다. 초기에는 수동 배포이지만, 나중에 여건이 되면 단계적으로 **배포 자동화 도구**를 붙여나가면 됩니다.

유지보수 측면에서도 이 접근이 유리합니다. 브랜치와 배포 이력이 명확히 구분되므로, **어떤 기능이 어떤 배포에 포함됐는지 추적**하기 쉽습니다. 문제가 생겨도 Git 이력을 통해 원인을 파악하고 필요한 부분만 수정하여 재배포하기 수월합니다. 또한 팀원들이 동일한 절차를 따르게 되므로, **새로운 인력이 참여해도 일정한 프로세스**에 따라 개발/배포할 수 있어 지식전파도 쉬워집니다.

결론적으로, **"Git 브랜치 전략 + 수동 배포 프로세스 개선"**이 현재 환경에서 가장 현실적입니다. 구체적으로 **Git Flow 기반으로 브랜치를 운용**하고, **필요시 cherry-pick이나 revert로 유연하게 대응**하며, **태그와 문서화로 배포 내역을 관리**하는 방안을 추천드립니다. 이와 함께 가능하다면 Jenkins 등의 **무료 CI 도구**를 도입해 빌드 단계의 오류를 줄이고, 향후 **유료 솔루션이나 자동화**를 도입할 수 있게 기반을 마련해두면 금상첨화입니다. 이러한 전략은 **운영 반영 오류를 최소화**하면서도 현재의 수동 배포 제약 안에서 실행 가능하며, 추후 환경이 개선될 때 **무리 없이 확장**해 나갈 수 있을 것입니다. [28] [18]

---

1 4 5 24 25 26 28 Git Branching strategy idea and git flow customization - Stack Overflow
https://stackoverflow.com/questions/58823641/git-branching-strategy-idea-and-git-flow-customization

2 6 7 8 11 12 13 14 27 Gitflow Workflow | Atlassian Git Tutorial
https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

3 9 10 [GIT] 깃 브랜치 전략 정리 - Github Flow / Git Flow
https://inpa.tistory.com/entry/GIT-%E2%9A%A1%EF%B8%8F-github-flow-git-flow-%F0%9F%93%88-%EB%B8%8C%EB%9E%9C%EC%B9%98-%EC%A0%84%EB%9E%B5

15 Jenkins
https://www.jenkins.io/

16 GitHub Actions billing - GitHub Docs
https://docs.github.com/billing/managing-billing-for-github-actions/about-billing-for-github-actions

17 GitHub - togglz/togglz: Feature Flags for the Java platform
https://github.com/togglz/togglz

18 Continuous Deployment & Delivery Software for DevOps teams | Octopus Deploy - Octopus Deploy
https://octopus.com/

19 version control - Git Flow process to send features for testing, only deploy specific feature to live - Stack Overflow
https://stackoverflow.com/questions/31338771/git-flow-process-to-send-features-for-testing-only-deploy-specific-feature-to-l

20 22 release management - Releasing untested features using git flow - Software Engineering Stack Exchange
https://softwareengineering.stackexchange.com/questions/442065/releasing-untested-features-using-git-flow

21 Git Feature Branch Workflow - GitHub Gist
https://gist.github.com/forest/19fc774dde34f77e2540

23 git - Branching strategy for releasing only approved/tested code - Stack Overflow
https://stackoverflow.com/questions/78570657/branching-strategy-for-releasing-only-approved-tested-code