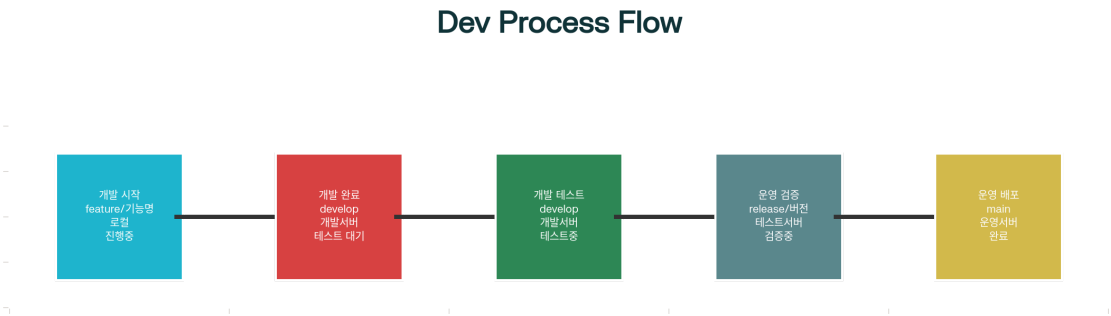
GitHub Desktop을 활용한 실전 소스/배포 관리 프로세스 개선안

현재 2명이 진행하는 프로젝트에서 겪고 있는 문제점들을 분석해서, GitHub Desktop만으로도 안전하고 효율적으로 협업할 수 있는 구체적인 해결책을 제시해드릴게요.



GitHub Desktop 기반 개선된 브랜치 전략 흐름도

현재 프로세스의 문제점 분석

주요 위험 요소들

테스트와 운영 코드 혼재 문제: 개발서버에 모든 기능이 섞여서 올라가다 보니, 운영에 반영하지 않을 기능까지 포함되어 서비스 오작동이 발생하고 있어요[1] [2] [3].

수동 소스 추리기의 한계: 개발자가 직접 필요한 소스만 골라서 복사 붙여넣기 하는 과정에서 실수가 발생하고, 이게 민원으로 이어지고 있죠[4] [5].

협업 충돌 관리 부재: 2명이 동시에 작업할 때 어떤 순서로 어떻게 병합할지에 대한 명확한 규칙이 없어서 충돌 상황에서 당황하게 됩니다[6] [7].

개선된 브랜치 전략 및 워크플로우

브랜치 구조 재정의

main 브랜치: 운영서버와 완전히 동일한 안정된 코드만 보관
develop 브랜치: 개발서버 배포용, 테스트를 위한 통합 브랜치
feature 브랜치: 개별 기능 개발용 (feature/기능명 형식)
release 브랜치: 운영 배포 직전 최종 검증용 (release/버전명 형식)

단계별 워크플로우

1단계: 개발 시작

```
1. develop 브랜치에서 최신 코드 pull
2. GitHub Desktop에서 "Current Branch" → "New Branch" 클릭
3. feature/기능명 형태로 브랜치 생성 (예: feature/login-api)
4. "Publish branch"로 원격에 브랜치 생성
```

2단계: 개발 진행

```
1. STS4에서 개발 작업
2. GitHub Desktop에서 변경사항 확인
3. 의미있는 단위로 커밋 (한글로 명확히 작성)
4. 정기적으로 Push origin 실행
```

3단계: 개발 완료 후 통합

```
1. feature 브랜치에서 develop으로 Pull Request 생성
2. 상대방 개발자가 코드 리뷰 진행
3. 충돌 발생시 Visual Studio Code에서 해결
4. 리뷰 완료 후 develop에 merge
5. 개발서버에 develop 브랜치로 WAR 파일 배포
```

4단계: 운영 배포 준비

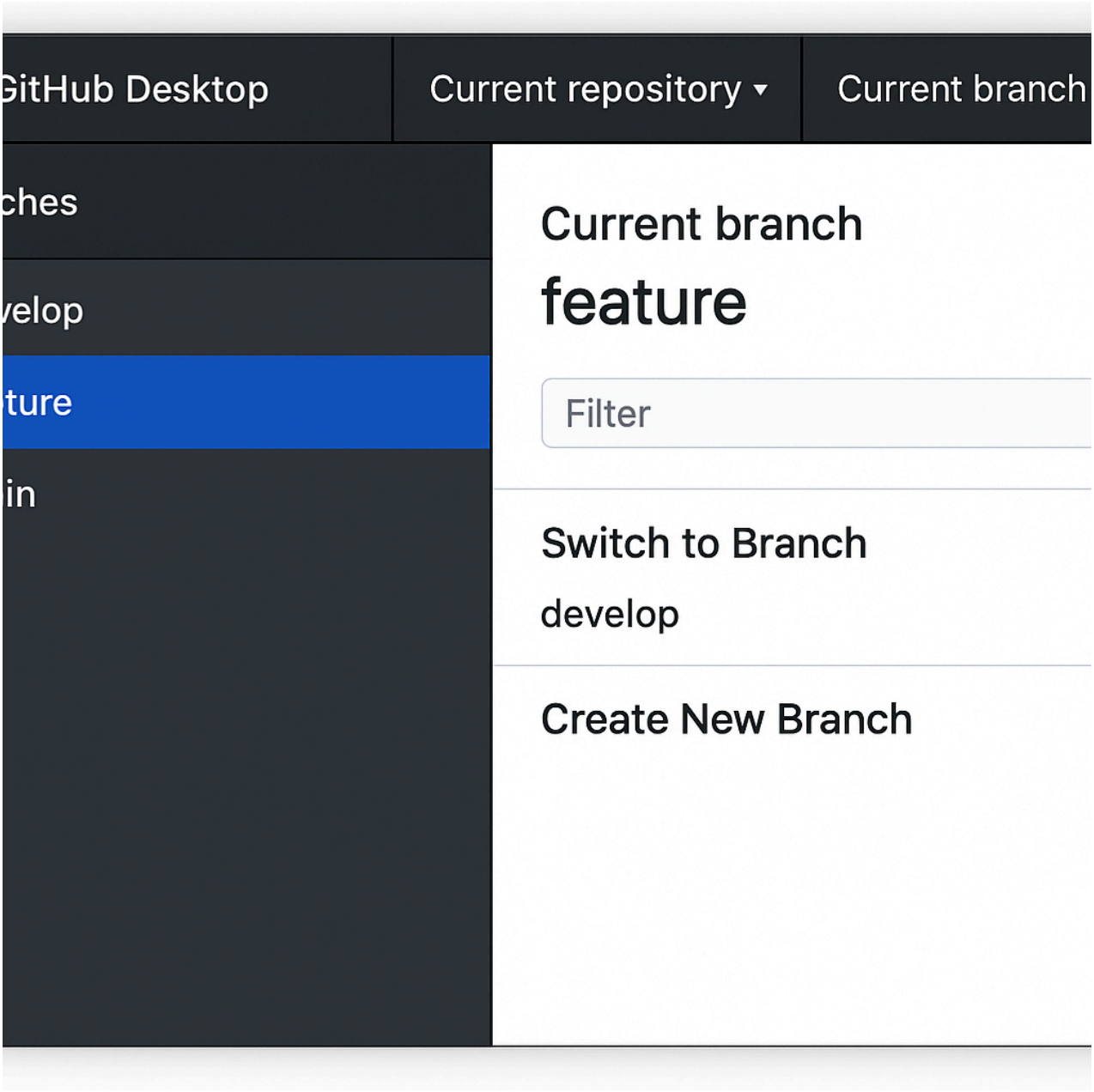
```
1. 클라이언트가 "운영 반영" 요청한 기능들만 선별
2. develop에서 release/v1.0 브랜치 생성
3. 필요한 기능만 cherry-pick으로 선택적 포함
4. 테스트서버에서 최종 검증
```

5단계: 운영 배포

```
1. release 브랜치에서 main으로 Pull Request
2. 최종 검토 후 merge
3. main 브랜치로 운영서버 WAR 파일 배포
4. 태그 생성으로 버전 관리
```

GitHub Desktop 실전 매뉴얼

브랜치 생성 및 전환



GitHub Desktop 브랜치 관리 화면 예시

새 브랜치 생성하기:

1. 상단의 "Current Branch" 버튼 클릭
2. "New Branch" 선택
3. 브랜치명 입력 (feature/기능명 형식 권장)
4. "Create Branch" 클릭
5. "Publish Branch"로 원격에 생성

브랜치 간 이동하기:

1. "Current Branch" 클릭
2. 목록에서 원하는 브랜치 선택
3. 자동으로 해당 브랜치로 체크아웃

충돌 해결 프로세스

충돌 발생 시 대응법[8] [9] [10]:

1. GitHub Desktop에서 충돌 알림 확인
2. "Open in Visual Studio Code" 클릭
3. 충돌 파일에서 `<<<<<<<`, `=======`, `>>>>>>>` 마커 확인
4. 필요한 코드만 남기고 마커 모두 삭제
5. 저장 후 GitHub Desktop으로 돌아가서 "Continue merge" 클릭

커밋 및 푸시 모범 사례

효과적인 커밋 메시지 작성:

```
좋은 예: "로그인 API 추가 - JWT 토큰 발급 기능 구현"
나쁜 예: "수정", "test", "asdf"
```

커밋 단위 조절:

- 하나의 기능 완성 시점에 커밋
- 너무 작은 단위로 쪼개지 말기
- 관련 없는 변경사항은 별도 커밋으로 분리

롤백 및 복구 전략

잘못된 커밋 되돌리기[11] [12]:

1. History 탭에서 되돌릴 커밋 우클릭
2. "Revert changes in commit" 선택
3. 새로운 revert 커밋 생성됨
4. Push origin으로 반영

파일 복구하기[13]:

1. History에서 파일이 존재했던 커밋 찾기
2. 해당 커밋에서 파일 우클릭
3. "View in Explorer"로 파일 확인 후 복사

소스 혼입 방지 및 안전 배포 전략

cherry-pick을 활용한 선별적 배포

운영 반영 기능만 선별하는 방법:

1. Git Bash에서 release 브랜치로 이동
2. `git cherry-pick <커밋해시>` 명령으로 필요한 커밋만 가져오기
3. 여러 커밋의 경우 `git cherry-pick A..Z` 범위 지정 가능

환경별 WAR 파일 관리

개발서버 배포 (자동화):

```
# STS4에서 Maven build
1. 프로젝트 우클릭 → Run As → Maven build
2. Goals에 "clean package" 입력
3. target 폴더에서 WAR 파일 생성
4. DEV 톰캣 webapp 폴더에 복사
```

운영서버 배포 (수동 검증):

```
# 체크리스트 기반 배포
1. release 브랜치에서 WAR 빌드
2. 테스트 환경에서 기능 검증
3. 클라이언트 승인 후 운영 배포
4. 배포 후 모니터링
```

협업 시 충돌 예방 및 해결책

GitHub Desktop 협업 시 주요 문제점과 해결 방안

충돌 최소화 전략

작업 영역 분리[14] [15]:

- 각자 담당할 패키지/폴더 미리 분할
- 공통 파일(config, util 등) 수정 시 사전 협의
- 일일 sync-up으로 작업 현황 공유

정기적인 동기화:

```
아침: develop 브랜치 pull로 최신 상태 유지
점심: 진행 상황 공유 및 충돌 가능성 체크
퇴근: 당일 작업분 commit & push
```

문제 상황별 대응 매뉴얼

상황 1: 동시에 같은 파일 수정

- GitHub Desktop에서 충돌 감지 시 즉시 상대방과 소통
- VS Code의 merge conflict 도구로 해결
- 해결 후 반드시 테스트 실행

상황 2: 실수로 잘못된 브랜치에 작업

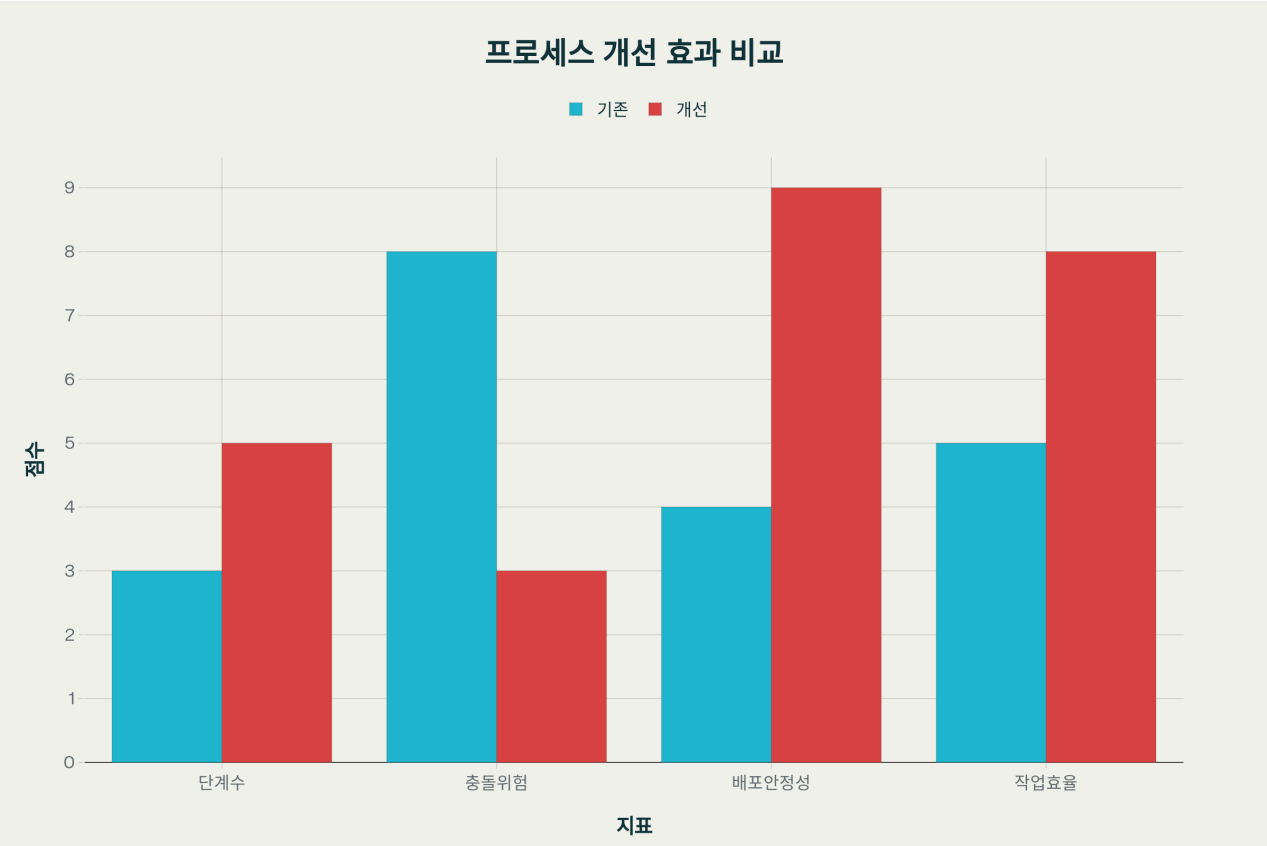
- GitHub Desktop의 "Cherry-pick" 기능으로 커밋 이동
- 원래 브랜치에서 해당 커밋 제거
- 올바른 브랜치에서 작업 재개

상황 3: 원격과 로컬 상태 불일치

- "Fetch origin"으로 원격 상태 확인
- 필요시 "Pull origin"으로 최신화
- 로컬 변경사항과 충돌 시 stash 활용

비효율 요소 개선 방안

현재 프로세스 vs 개선된 프로세스



기존 vs 개선된 GitHub Desktop 프로세스 비교

자동화 없이 가능한 개선사항

브랜치 전략 표준화:

```
main (운영) ← release (운영 검증) ← develop (개발 통합) ← feature (개발 작업)
```

체크리스트 도입:

```
## 개발 완료 체크리스트
- [ ] 기능 테스트 완료
- [ ] 코드 리뷰 요청
- [ ] 충돌 해결 확인
- [ ] develop 브랜치에 merge

## 운영 배포 체크리스트
- [ ] 클라이언트 승인 완료
- [ ] release 브랜치에서 테스트
- [ ] WAR 파일 백업
- [ ] 운영 배포 및 모니터링
```

문서화 및 히스토리 관리[16] [17]:

- 각 배포별 변경사항 문서화
- Git 태그로 버전 관리
- 배포 로그 및 롤백 계획 수립

팀 차원의 권장 사항

역할 분담

개발자 A: 백엔드 API, 데이터베이스 관련 작업
개발자 B: 프론트엔드 UI, 화면 관련 작업
공통: 설정 파일, 공통 모듈은 사전 협의 후 작업

소통 규칙

일일 스탠드업: 작업 계획 및 충돌 가능성 미리 체크
코드 리뷰: 모든 PR은 상대방 리뷰 후 merge
긴급 상황: 운영 이슈 발생 시 즉시 소통 및 핫픽스 진행

학습 및 개선

GitHub Desktop 숙련도 향상[18] [19] [20]:

- 매주 한 번씩 새로운 기능 익히기
- 충돌 해결, 브랜치 관리 등 실전 연습
- 문제 상황 발생 시 해결 과정 문서화

이렇게 체계적으로 프로세스를 개선하면, 현재 겪고 있는 소스 혼입, 배포 실수, 협업 충돌 문제들을 크게 줄일 수 있을 거예요. GitHub Desktop만으로도 충분히 안전하고 효율적인 협업이 가능합니다!

⁂

1. https://lee-ding.tistory.com/117
2. https://www.memoengine.com/blog/github-desktop-설치-및-사용/
3. https://wonsss.github.io/etc/git-branch-strategy/

4. https://dev-hyonie.tistory.com/31
5. https://ossam5.tistory.com/611
6. https://adjh54.tistory.com/364
7. https://velog.io/@klolarion/Github-브랜치-관리
8. https://engineering-skcc.github.io/github pages/github-pages-desktop/
9. https://tecoble.techcourse.co.kr/post/2021-07-15-git-branch/
10. https://docs.github.com/ko/desktop/making-changes-in-a-branch/managing-branches-in-github-desktop
11. https://codegear.tistory.com/37
12. https://dev-district.tistory.com/23
13. https://staticclass.tistory.com/130
14. https://read-me.tistory.com/entry/Git-Github-desktop-설치-및-사용법
15. https://blog.hwahae.co.kr/all/tech/14184
16. https://eunyoe.tistory.com/210
17. https://velog.io/@lazysia/git-github-desktop-사용하기
18. https://velog.io/@hxeyexn/Git-Branch-Strategy
19. https://best-study-day.tistory.com/6
20. https://www.youtube.com/watch?v=0YsMEPxi_wc