BOOTS ON THE GROUND AI

AI Solutions for Small Business Owners

GITHUB PLAYBOOK

Never Push Without a Plan Again

Your complete guide to GitHub for Next.js & React web apps.
Rules, checklists, workflows, and templates that ensure
every push is clean, reviewed, and production-ready.

Built for builders who want version control done right.

BOOTS APPROVED

Version 1.0 | February 12, 2026
Boots On The Ground AI | bootsagentai.com

TABLE OF CONTENTS

Your roadmap to GitHub mastery

01 WHY GITHUB?

The Boots Philosophy on Version Control

Version control is not optional. It's not a nice-to-have. It's the foundation of professional software development. At Boots On The Ground AI, we believe every line of code deserves to be tracked, reviewed, and protected.

BOOTS SAYS:

BOOTS SAYS: Think of GitHub like a time machine for your code. Every change is recorded. You can see who changed what, when, and why. If something breaks, you revert. If you need to collaborate, GitHub makes it seamless. It's the safety net for every developer.

Without GitHub, you face:

X No history - you cannot undo mistakes or understand why code changed

X Team chaos - multiple people editing the same file causes merge disasters

X No code review - bugs slip through because nobody reviewed the code

X Deployment roulette - you push hoping nothing breaks, with no rollback

X Lost knowledge - when a developer leaves, their context leaves with them

With GitHub, you get:

+ Complete history - see every change, who made it, and why

+ Safe collaboration - branches let teams work in parallel without conflicts

+ Code review - Pull Requests force good practices and catch bugs early

+ Instant rollback - deploy with confidence; revert instantly if needed

+ Automated testing - GitHub Actions run tests on every push automatically

THE BOOTS STANDARD:

THE BOOTS STANDARD: Every project at Boots On The Ground AI ships with a GitHub repo, proper .gitignore, and branch protection rules. No exceptions. If it does not have version control, it does not leave the shop.

02 THE 10 GOLDEN RULES OF GITHUB

Memorize these. Live by them. Never break them.

1 Never Work Directly on Main

Main branch is production. Always create a feature branch, make changes, and submit a Pull Request. Protect main with branch rules. No exceptions, no matter how small the change.

2 Write Meaningful Commit Messages

Commit messages must explain WHY, not WHAT. Bad: 'fixed bug'. Good: 'Fix login timeout by increasing session TTL to 24h'. Future you will thank present you.

3 Pull Before You Push

Always 'git pull' before 'git push'. Someone else may have pushed changes while you worked. Pulling first prevents conflicts and keeps history clean.

4 Use .gitignore from Day One

Create a .gitignore immediately. Exclude node_modules, .env, build artifacts, IDE files, OS files. Never commit secrets, passwords, or API keys. Use environment variables instead.

5 One Feature Per Branch

Each branch solves ONE problem. One feature, one bug fix, one refactor. Not five mixed changes. This makes code review faster, merges cleaner, and rollbacks safer.

6 Review Before You Merge

Every merge happens through a Pull Request. Pull Requests force code review, testing, and discussion. No pushing directly to main. One pair of eyes catches bugs you missed.

7 Tag Your Releases

Use semantic versioning for every release: v1.0.0, v1.0.1, v1.1.0, v2.0.0. Tags let you instantly deploy any version, communicate what changed, and manage dependencies.

8 Protect Your Main Branch

Enable branch protection rules: require Pull Request reviews, require status checks pass, require branches to be up to date. Make it hard to break production.

9 Use GitHub Actions for CI/CD

Automate tests on every push. Automate deploys on merge to main. GitHub Actions runs your tests, linting, and deployments. Failures block merges. Good code always ships.

10 Document Everything in README

Your README must explain: what the project does, how to set it up, how to run it, how to test it, and how to contribute. A new developer should clone, read the README, and be productive.

03 REPOSITORY STRUCTURE

The anatomy of a well-organized GitHub repo

Every repo should follow this structure. New developers should immediately understand where everything lives.

Directory/File	Purpose
.github/workflows/	CI/CD action files - automated testing and deployment
.github/PULL_REQUEST_TEMPLATE.md	Template for new Pull Requests - ensures consistency
.github/ISSUE_TEMPLATE/	Issue templates - bug reports, features, documentation
.gitignore	Files to exclude from tracking - dependencies, builds, secrets
README.md	Project documentation - setup, usage, contribution guide
LICENSE	Legal protection - MIT, Apache, or your choice
CONTRIBUTING.md	How others contribute - code style, PR process, testing
CHANGELOG.md	Version history - what changed in each release
src/	Source code - your Next.js, React, backend code lives here
tests/	Test files - unit tests, integration tests, E2E tests
docs/	Additional documentation - architecture, guides, API docs
package.json / requirements.txt	Dependencies - what packages your project needs

BOOTS EXPLAINS:

BOOTS EXPLAINS: The .github/ folder is your repo's command center. Put your CI/CD workflows there. They automatically run on every push and Pull Request. Tests fail? The merge is blocked. Code review is required? Cannot merge without approval. This folder automates your quality standards.

04 THE .GITIGNORE

Annotated Template for Next.js

Create a .gitignore file in your repo root on day one. Add entries as you discover files that should not be tracked.

# Dependencies (NEVER commit these - huge, regenerated via npm/yarn)

```
node_modules/

.pnp

.pnp.js
```

# Build outputs (Regenerated on every build)

```
.next/

out/

build/

dist/
```

# Environment variables (CRITICAL: Never commit real secrets here)

```
.env

.env.local

.env.*.local
```

# IDE and editor files (Personal preferences, not project code)

```
.vscode/

.idea/

*.swp

*.swo

*~

.DS_Store
```

# Testing (Coverage reports and temp test files)

```
coverage/

.nyc_output/
```

# Misc

```
.cache/

tmp/

temp/

*.log
```

DANGER:

CRITICAL: Never commit secrets to GitHub. Use environment variables and .env files. If you accidentally push secrets, assume they are compromised. Rotate them immediately. Use GitHub's secret scanning to catch mistakes.

05 GITHUB ACTIONS

CI/CD Template for Next.js

Save this as **.github/workflows/ci.yml**. This workflow runs on every push and Pull Request, automating tests and builds.

```
name: CI

on:
push:
branches: [main, develop]
pull_request:
branches: [main, develop]

jobs:
build:
runs-on: ubuntu-latest
strategy:
matrix:
node-version: [18.x, 20.x]

steps:
- uses: actions/checkout@v3

- name: Use Node.js ${{ matrix.node-version }}
uses: actions/setup-node@v3
with:
node-version: ${{ matrix.node-version }}
cache: 'npm'

- name: Install dependencies
run: npm ci

- name: Run linter
run: npm run lint
```

```
- name: Run tests

run: npm run test


- name: Build project

run: npm run build


- name: Upload coverage

uses: codecov/codecov-action@v3
```

BOOTS EXPLAINS:

BOOTS EXPLAINS: This workflow runs on every push and PR. It installs dependencies, runs linting, tests the code, and builds the project. If any step fails, the PR is blocked from merging. This guarantees that broken code never reaches main. Add more steps (security scanning, type checking, E2E tests) as needed.

06 BRANCHING STRATEGY

How to organize your code safely

Branch Naming Convention:

main - Production code. Only merge tested, reviewed code here.

develop - Staging area. Where features are integrated before production.

feature/feature-name - A new feature. Create from develop.

bugfix/bug-name - A bug fix. Create from develop.

hotfix/critical-issue - Emergency fix for production. Create from main.

Workflow:

Pull latest: `git pull origin develop`

Create branch: `git checkout -b feature/user-auth`

Make changes, commit often with meaningful messages

Push branch: `git push origin feature/user-auth`

Create Pull Request on GitHub (describe changes and tag reviewers)

Wait for code review and CI/CD checks to pass

Address review feedback with new commits

Once approved, merge via GitHub (not via command line)

Delete branch after merge

DANGER:

IMPORTANT: Never commit .env files. Use environment variables managed by your deployment platform (Vercel, Heroku, AWS). If you use a .env.local for development, add it to .gitignore. Document required variables in .env.example.

07 DAILY WORKFLOW

Your step-by-step guide to a productive day

1. **Start Your Day**
   Pull the latest code from remote: git pull origin main

2. **Create Your Branch**
   Create a feature branch from develop: git checkout -b feature/your-feature

3. **Write Code and Commit Often**
   Write a small piece of code. Test it. Commit with a meaningful message: git commit -m 'Add user login validation'

4. **Push Your Branch**
   Push to remote: git push origin feature/your-feature

5. **Create a Pull Request**
   Go to GitHub. Click 'Create Pull Request'. Describe your changes. Request review from teammates.

6. **Get Code Review**
   Wait for teammates to review. Address feedback. Push new commits to the same branch.

7. **Merge and Clean Up**
   Once approved, merge via GitHub. Delete the feature branch: git branch -d feature/your-feature

08 PRODUCTION DEPLOYMENT CHECKLIST

Before you push to production

Repository Setup

- [ ] README.md explains project and setup
- [ ] .gitignore excludes secrets and build artifacts
- [ ] LICENSE file is present
- [ ] CONTRIBUTING.md documents PR process

Branch Protection

- [ ] Main branch is protected
- [ ] Require Pull Request reviews (minimum 1)
- [ ] Require status checks to pass (CI/CD)
- [ ] Require branches up to date before merge
- [ ] Dismiss stale Pull Request reviews

CI/CD

- [ ] GitHub Actions workflow runs on every PR
- [ ] Tests pass in CI/CD
- [ ] Build succeeds in CI/CD
- [ ] Code coverage meets minimum threshold
- [ ] Linter passes without warnings

Security

- [ ] No API keys or secrets in code
- [ ] Environment variables for all config
- [ ] Dependencies are up to date
- [ ] No vulnerable packages (npm audit)
- [ ] GitHub secret scanning is enabled

Documentation

- ☐ Code has comments explaining complex logic
- ☐ README has deployment instructions
- ☐ CHANGELOG is updated with changes
- ☐ API endpoints are documented

09 COMMAND QUICK REFERENCE

Copy-paste your way to git mastery

BASICS

Command	What it does
`git clone <url>`	Clone a repo to your machine
`git status`	See what files changed
`git add <file>`	Stage a file for commit
`git add .`	Stage all changed files
`git commit -m "message"`	Commit staged changes
`git push`	Push commits to remote
`git pull`	Pull latest from remote

BRANCHING

Command	What it does
`git checkout -b feature/name`	Create and switch to new branch
`git branch`	List local branches
`git branch -a`	List all branches (local + remote)
`git checkout main`	Switch to main branch
`git branch -d feature/name`	Delete a local branch
`git push origin feature/name`	Push branch to remote

UNDOING CHANGES

Command	What it does
`git restore <file>`	Discard changes in a file
`git reset HEAD~1`	Undo last commit (keep changes)
`git revert <commit>`	Create new commit that undoes changes
`git log`	See commit history
`git show <commit>`	See what changed in a commit

10 TROUBLESHOOTING

Solutions to common GitHub problems

Merge Conflicts

Two branches changed the same file. Git cannot auto-merge. Run 'git status' to see conflicted files. Edit each file, remove the conflict markers (<<<, >>>, ===). Then 'git add' and 'git commit'.

Accidentally Committed to Main

You made a commit on main instead of a branch. Run 'git reset HEAD~1' to undo. Create a branch 'git checkout -b feature/name'. Commit again. Now create a PR. Never force push main.

Need to Undo Last Commit

Committed but not pushed? Run 'git reset HEAD~1' (keeps changes) or 'git reset --hard HEAD~1' (discards changes). Already pushed? Use 'git revert ' to create a new commit that undoes it.

Pushed Secrets to GitHub

EMERGENCY. Your secrets are compromised. Rotate them immediately (new API keys, passwords, tokens). Remove the file: 'git rm --cached .env' and commit. The file still exists in history but is no longer tracked. Use github.com/gitleaks or tools to scan.

Large File Blocking Push

You committed a large file (video, archive, binary). Git refuses to push. Remove it: 'git rm --cached '. Add to .gitignore. Commit. For legitimate large files, use Git LFS (Large File Storage).

Permission Denied

SSH key not set up. Run 'ssh-keygen -t ed25519'. Copy the public key to GitHub Settings > SSH Keys. Test with 'ssh -T git@github.com'.

Detached HEAD State

You checked out a specific commit instead of a branch. You are not on a branch anymore. Run 'git checkout main' to return to main.

.gitignore Not Working

You added a file to .gitignore AFTER committing it. Git still tracks it. Run 'git rm --cached ' then commit. Now Git will ignore it.

11 PRE-BUILD CHECKLIST

Tear this out. Check it before every deployment.

Project Setup

- ☐ Repository exists and is public/private as needed
- ☐ README.md is complete and clear
- ☐ LICENSE is chosen and added
- ☐ CONTRIBUTING.md explains how to contribute

Security

- ☐ No secrets in code (API keys, passwords, tokens)
- ☐ All sensitive data in environment variables
- ☐ .env files in .gitignore
- ☐ GitHub secret scanning is enabled
- ☐ Dependencies scanned for vulnerabilities

CI/CD

- ☐ .github/workflows/ contains CI/CD configurations
- ☐ Tests run on every push and PR
- ☐ Build succeeds in CI/CD environment
- ☐ Code coverage meets minimum threshold
- ☐ Linting passes without errors

Documentation

- ☐ README has setup instructions
- ☐ README has usage examples
- ☐ API endpoints documented
- ☐ Environment variables documented
- ☐ CHANGELOG updated with version history

THE BOOTS STANDARD:

THE BOOTS STANDARD: Add GitHub Actions workflows to .github/workflows/. Automate testing, linting, and deployments. Your CI/CD should be bulletproof. Code that does not pass tests cannot merge. Good code always ships.