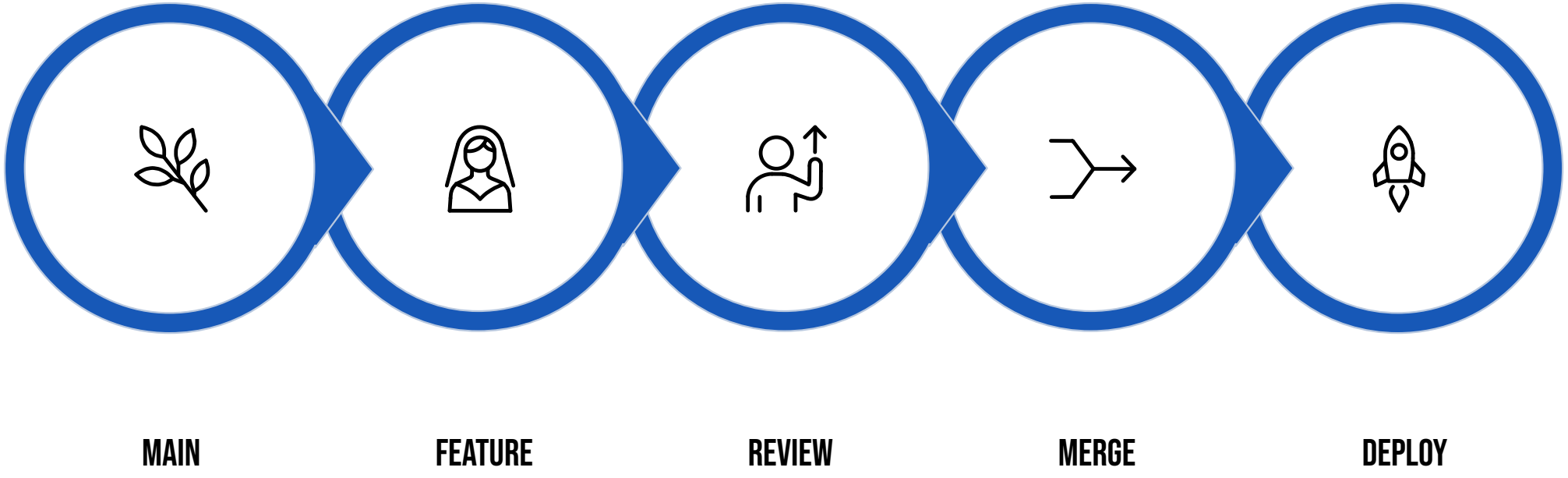
SESSION 8: GIT BRANCHING, MERGING & CONFLICT RESOLUTION

 **by Oluwatobiloba Durodola**

SESSION 8: GIT BRANCHING, MERGING & CONFLICT RESOLUTION

MASTERING TEAM COLLABORATION WITHOUT BREAKING CODE

This session aims to transition you from a solo Git user to a team-ready collaborator. We'll cover how to effectively manage code with branches, seamlessly integrate changes through merges, and resolve conflicts like a pro.



It's time to **branch out** and master these essential skills for efficient team development!

WHY BRANCHING MATTERS IN REAL TEAMS

Git branching is more than just a feature; it's the foundation for effective team collaboration in modern software development. It allows multiple developers to work concurrently on different aspects of a project without interfering with each other's work.

→ ENABLES PARALLEL WORK

Teams can develop new features, implement urgent fixes, and run experiments simultaneously, preventing bottlenecks and maximizing productivity.

→ PROTECTS PRODUCTION

It ensures the main branch remains stable and deployable at all times, safeguarding your continuous integration/continuous deployment pipelines.

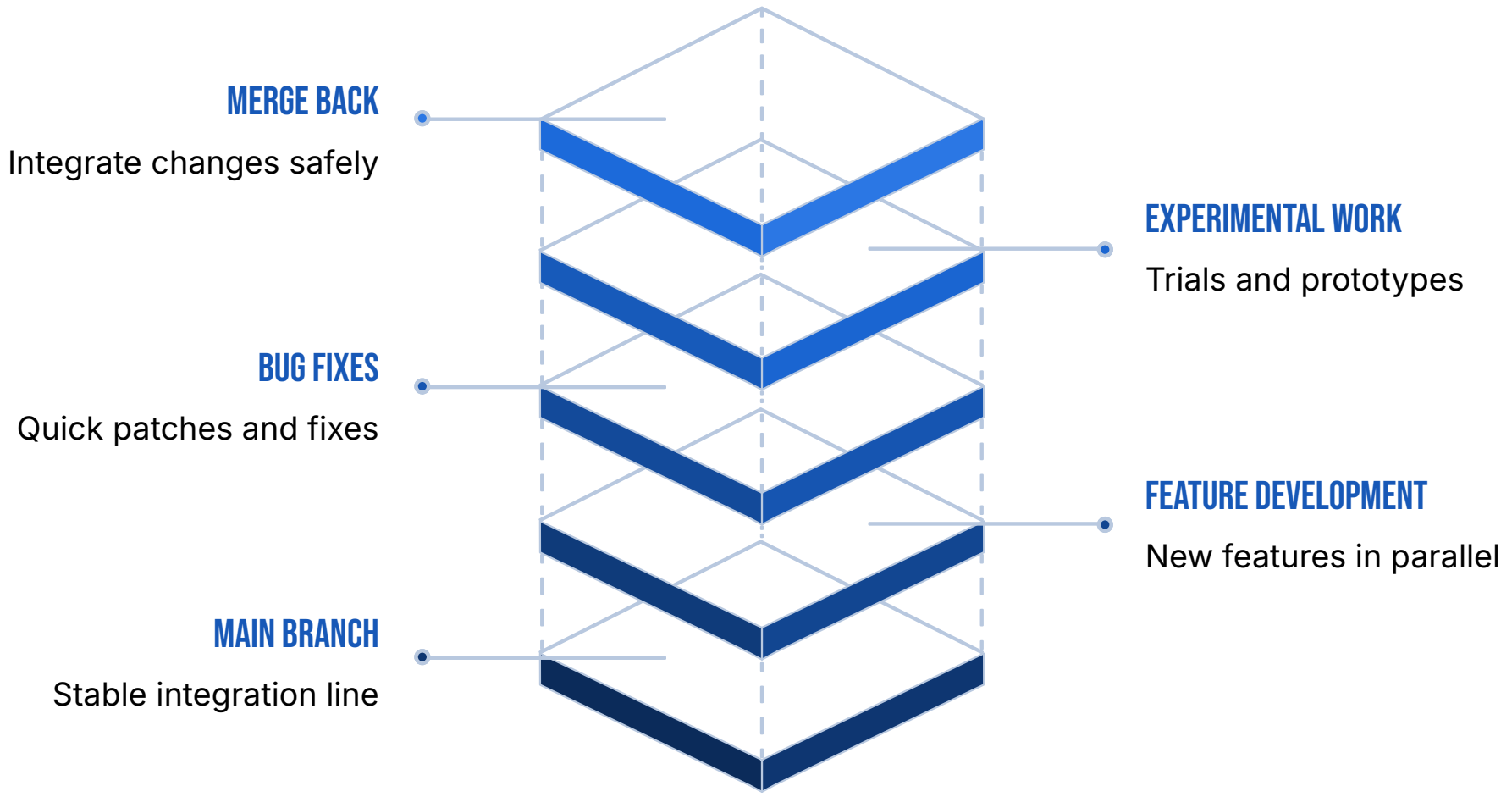
→ SUPPORTS MODERN WORKFLOWS

Branching is crucial for implementing essential practices like pull requests, comprehensive code reviews, and agile development methodologies.

→ REDUCES RISK

Isolate potentially buggy or unfinished code on dedicated branches, preventing it from impacting the stable codebase or other team members.

Without branching, collaboration quickly devolves into a single-file nightmare. Branching keeps things organized, safe, and allows teams to innovate faster. 👉



WHAT IS A BRANCH? (DEEP DIVE)

At its core, a Git branch is simply a **lightweight, movable pointer to a commit**. It allows you to create an independent line of development, isolating changes from the main codebase until they are ready to be integrated.

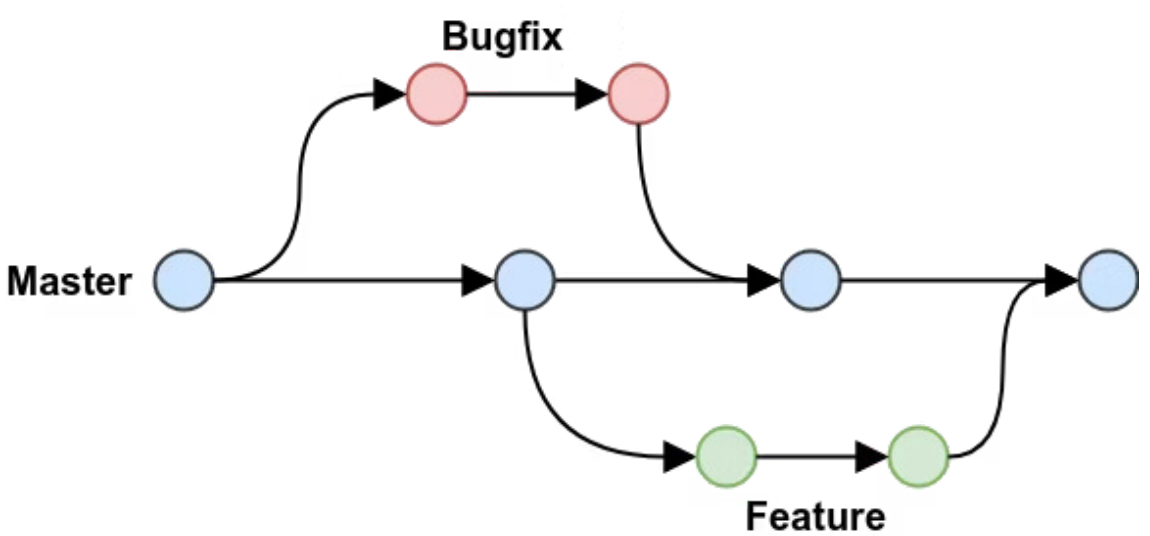
> ### THINK OF IT AS: A "SAVE POINT"
>
> Like a save point in a video game, branching allows you to experiment with new features or fixes without jeopardizing your current progress. You can always revert or switch back to a stable state.

> ### THINK OF IT AS: A FORK IN A RIVER
>
> Imagine a river that splits into two paths. Your main codebase is the river, and each branch is a separate path where changes flow independently until they eventually merge back together downstream.

The **default branch** in most modern repositories is named `"main"` (older repositories often used "master"). This branch typically represents the stable, deployable version of your project.

Advanced Note: Branches are incredibly cheap to create. Git doesn't duplicate all your files for each new branch; it simply creates a new pointer that references a specific commit in your repository's history, making branching a highly efficient operation.

VISUALIZING BRANCHING

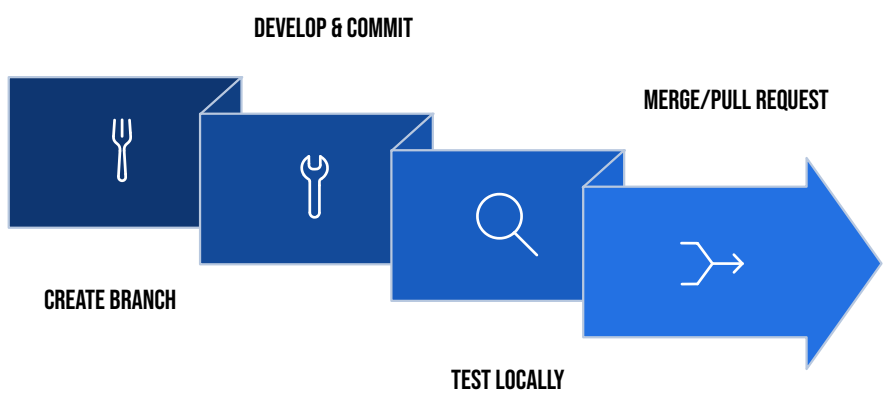


SIMPLE REPOSITORY HISTORY

```
* (HEAD -> feature-x) Add new feature logic
* Update UI for feature X
| * (main) Merge feature-x into main
| |\
| | * (feature-x) Implement basic feature
| |/
| * Fix critical bug in authentication
|/
* Initial commit and project setup
```

BRANCHING WORKFLOW (BIG PICTURE + VARIATIONS)

Understanding the standard branching workflow and its common variations is key to efficient and collaborative Git usage. This section outlines the typical steps and introduces how different branching strategies cater to specific development needs.



WORKFLOW VARIATIONS

- **Feature Branching:** Ideal for developing new features. Each feature gets its own branch, keeping the main codebase clean and stable.
- **Release Branching:** Used to prepare for a new release. Bug fixes are applied here, and once stable, it's merged into main and tagged.

These strategies allow teams to manage complexity, isolate experimental changes, and maintain high code quality.

KEY GIT COMMANDS

Here are the fundamental commands you'll use in these workflows:

- `git branch [branch-name]`: Create a new branch.
- `git checkout -b [branch-name]`: Create and switch to a new branch.
- `git merge [branch-name]`: Integrate changes from one branch into another.
- `git pull`: Fetch from and integrate with another repository or a local branch.

> Mastering these commands forms the backbone of collaborative development in Git.

VIEWING AND MANAGING BRANCHES

Learn how to inspect your repository's branching structure and perform essential maintenance tasks to keep your codebase organized.

KEY BRANCH MANAGEMENT COMMANDS

Use these commands to navigate and control your branching environment:

- `git branch`: Lists all local branches, with an asterisk (*) indicating the currently active branch.
- `git branch -a`: Shows both local and remote-tracking branches, providing a complete view.
- `git branch -d [branch-name]`: Deletes a local branch that has already been successfully merged into its upstream branch.
- `git branch -D [branch-name]`: Forcibly deletes a local branch, even if it contains unmerged changes. Use with extreme caution!

Branches are local by default. They synchronize with remote repositories only when you explicitly use `git push` to send your changes or `git pull` to fetch updates.

VISUALIZING YOUR BRANCH HISTORY

For a concise and visual overview of your branches and their commit history, use the following command:

```
git log --graph --oneline --all
```

CREATING A BRANCH (BEST PRACTICES)

Creating a new branch in Git is a fundamental operation that allows you to isolate your work. Here's how to do it and what to keep in mind for effective team collaboration.

KEY COMMANDS

The most common way to create a new branch and immediately switch to it is:

```
git checkout -b feature/user-auth
```

Alternatively, you can create the branch first and then switch to it:

```
git branch new-branch
```

```
git checkout new-branch
```

Both commands effectively create a new pointer to your current commit, making it the starting point for your new, independent line of development.

BEST PRACTICES FOR BRANCHING

1

CLEAR NAMING CONVENTIONS

Adopt descriptive prefixes like `feature/`, `bugfix/`, or `hotfix/` followed by a concise, descriptive name (e.g., `feature/user-login-v2`). This improves readability and helps quickly identify the branch's purpose.

2

BASE FROM AN UP-TO-DATE `main`

Always create new feature or bugfix branches from an up-to-date `main` branch. This minimizes merge conflicts later on and ensures you're working with the latest stable codebase.

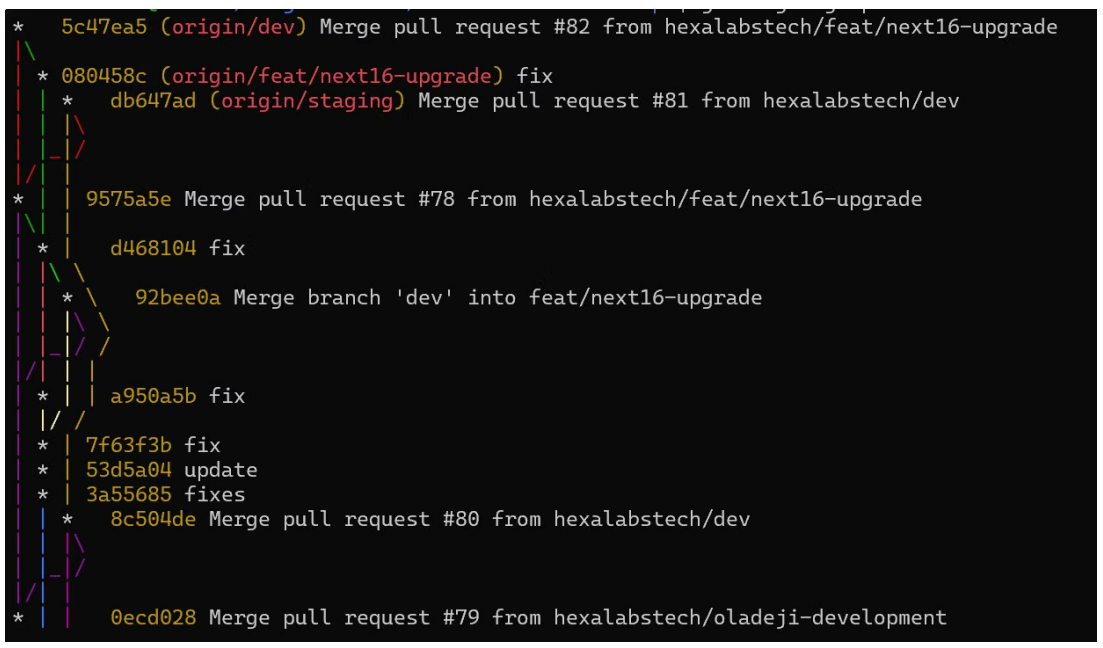
3

KEEP BRANCHES FOCUSED

Each branch should ideally address a single feature, bug, or task. Avoid cramming multiple, unrelated changes into one branch, which can complicate code reviews and integration.

VISUALIZING BRANCH CREATION

Observe how a new branch, `feature/user-auth`, diverges from the `main` branch, allowing for independent development.



SWITCHING BETWEEN BRANCHES

Switching branches is a core Git operation that allows you to move between different lines of development, updating your working directory to reflect the state of the chosen branch.

COMMANDS FOR BRANCH SWITCHING

To switch to an existing branch, use the `git checkout` command:

```
git checkout main
```

```
git checkout feature/user-auth
```

Alternatively, the more modern and often preferred command for switching branches (and creating new ones) is `git switch`:

```
git switch main
```

```
git switch feature/user-auth
```

When you switch branches, Git efficiently updates your working directory to match the content of the target branch's latest commit. This means files may appear, disappear, or change content to reflect the state of that branch.

IMPORTANT WARNINGS & BEST PRACTICES

STASH UNCOMMITTED CHANGES

If you have uncommitted changes in your working directory, Git might prevent you from switching branches to avoid overwriting your work. To handle this, use `git stash` to temporarily save your changes before switching, and `git stash pop` to restore them on the new branch (or the original one).

AVOID DETACHED HEAD STATE

Directly checking out a specific commit (e.g., `git checkout <commit-hash>`) will put you in a "detached HEAD" state. While useful for inspecting old versions, any new commits you make here won't belong to a branch and can be lost easily. Always create a new branch from a commit if you intend to continue working from that point.

MAKING CHANGES ON A BRANCH

Once you've created and switched to a new branch, you can begin making changes. The core workflow involves editing files, staging your changes, and committing them to your branch's history.

01

EDIT FILES

Modify your code, add new features, or fix bugs in your working directory.

02

STAGE CHANGES (`git add`)

Selectively add modified files or new files to the staging area using `git add .` or `git add [file]`, preparing them for the next commit.

03

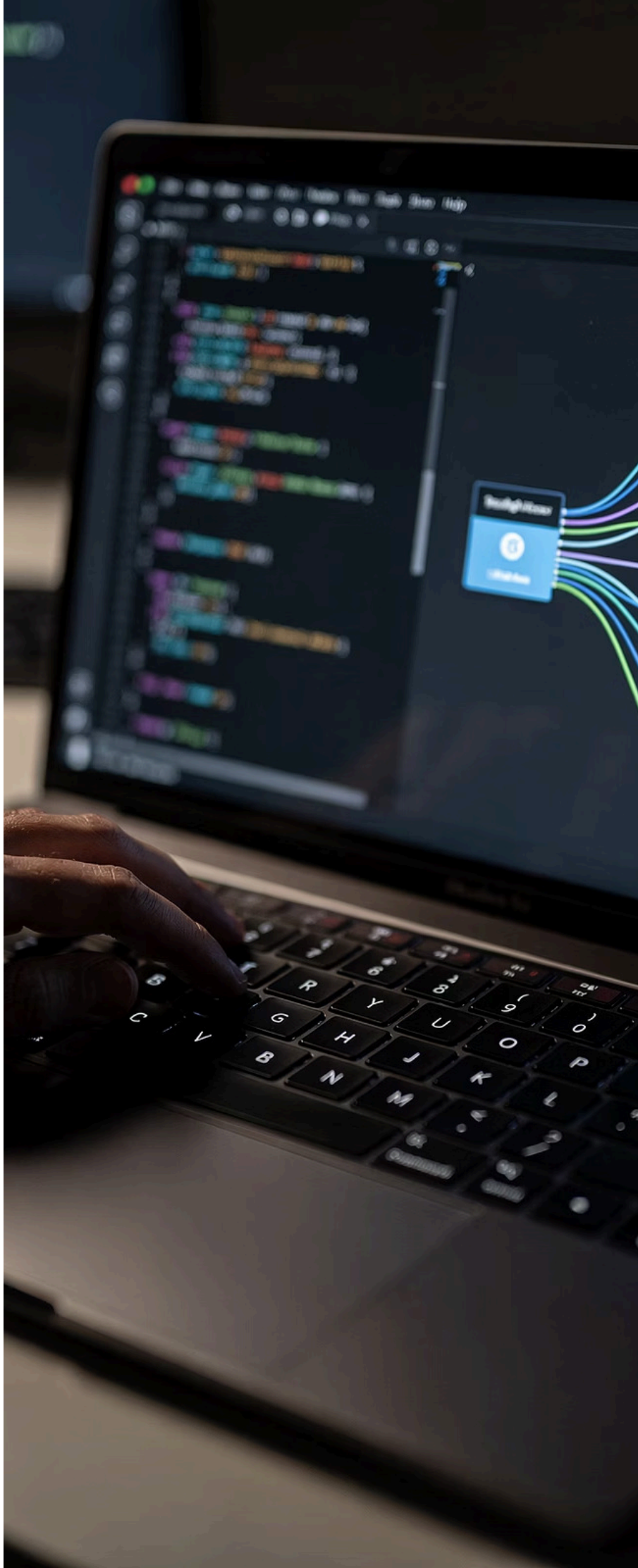
COMMIT CHANGES (`git commit`)

Record the staged changes as a new snapshot in your branch's history with a clear, descriptive message using `git commit -m "Descriptive message"`.

> **Important Reminder:** Commits are inherently branch-specific. This means the history you build on one branch develops independently from others until you explicitly merge them. This isolation is key for parallel development without interference.

ADVANCED TIP: ATOMIC COMMITS

For a cleaner and more manageable project history, always strive for **atomic commits**. An atomic commit encapsulates a single, logical change. This practice makes code reviews simpler, eases the process of reverting specific changes, and facilitates better understanding of project evolution.



PUSHING BRANCHES TO REMOTES

Pushing your local branches to a remote repository is essential for collaboration, sharing your work, and creating backups. This synchronizes your local changes with a central server, making them accessible to your team.

KEY COMMANDS

```
git push origin feature/user-auth
```

This command pushes your local `feature/user-auth` branch to the `origin` remote repository. If this is the first time pushing a new local branch, you'll need to set the upstream tracking branch:

```
git push -u origin branch-name
```

The `-u` (or `--set-upstream`) flag links your local branch to the remote one, simplifying future pushes and pulls.

WHY PUSH TO REMOTE?

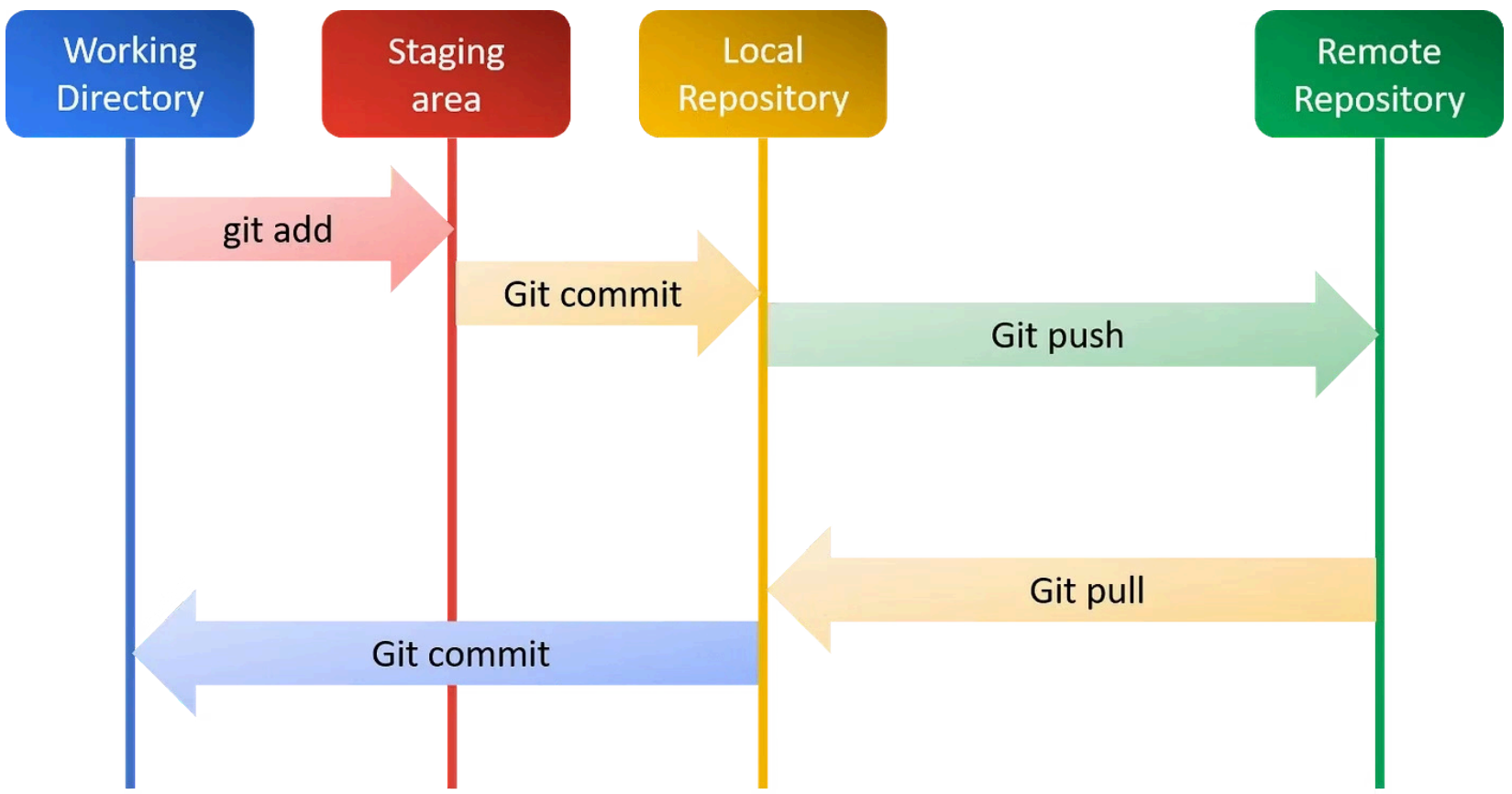
Pushing your branches allows teammates to review, collaborate on, and integrate your code. It also serves as a crucial backup, protecting your work from local data loss.

BEST PRACTICE: PULL BEFORE PUSH

Always pull the latest changes from the remote repository before pushing your own. This minimizes the risk of merge conflicts and ensures you're working with an up-to-date codebase.

VISUALIZING THE PUSH OPERATION

This diagram illustrates the flow of changes when pushing a local branch to a remote repository.



Merging Branches: Fast-Forward vs. True Merge

After developing features on an isolated branch, the next crucial step is to integrate those changes back into a main development line, typically your `main` branch. This process is called merging, and Git handles it in two primary ways depending on your commit history.

To merge a branch into your current branch (e.g., merging `feature/user-auth` into `main` while on `main`):

```
git merge feature/user-auth
```

Fast-Forward Merge

Occurs when the history of the branch being merged is linear with the target branch. Git simply moves the target branch's pointer forward to the latest commit of the merged branch, effectively integrating changes without creating a new commit. This is the default behavior when there are no divergent commits.

True Merge (3-Way Merge)

Happens when both branches have diverged, meaning there have been new commits on both the feature branch and the target branch since they split. Git creates a new "merge commit" that ties together the histories of both branches, preserving the individual branch histories.

You can force Git to always create a merge commit, even if a fast-forward merge is possible, by using the `--no-ff` flag. This explicitly preserves the history of your feature branches, which can be useful for maintaining a clear record of when feature work was integrated:

```
git merge --no-ff feature/user-auth
```

Visualizing the Merge Process

Image from nvie.com/posts/a-successful-git-branching-model/ by Vincent Driessen

Rebasing vs. Merging: Choosing Your Integration Strategy

Both rebasing and merging are ways to integrate changes from one Git branch into another. However, they achieve this in fundamentally different ways, impacting your project's commit history and collaboration workflow.

Rebasing

Rebasing involves moving or combining a sequence of commits to a new base commit. Essentially, it rewrites commit history by reapplying your branch's commits one by one on top of another base branch (e.g., `main`).

```
git rebase main
```

- **Pros:** Creates a clean, linear project history without "merge commits," making the history easier to read and follow.
- **Cons:** Rewrites commit history, which can be problematic if the rebased branch has already been pushed and shared with others. It can lead to confusion and necessitate force pushes.

When to use: Ideal for cleaning up local, unpushed feature branches before integrating them into a shared branch. It's often preferred for personal branches or during development phases where history can be streamlined.

Merging

Merging integrates a divergent line of development into your current branch by creating a new "merge commit." This merge commit has two parent commits (the tips of the two merged branches), preserving the complete history of both branches.

```
git merge feature-branch
```

- **Pros:** Preserves the exact history of how and when features were integrated, including the existence of feature branches. It never rewrites history, making it safe for shared branches.
- **Cons:** Can lead to a "messy" history with many merge commits, sometimes referred to as "merge bubbles," especially in busy repositories.

When to use: Best for integrating changes into shared branches (like `main` or `develop`) after features are complete, ensuring that the team's shared history remains intact and consistent.

Key Takeaway: Never Rebase Shared History

The golden rule of rebasing is: **never rebase commits that have already been pushed to a public repository and shared with others.** This can break other developers' local repositories and lead to significant headaches.

WHAT IS A MERGE CONFLICT?

A merge conflict occurs when Git attempts to combine two divergent histories, and both histories have made changes to the same part of the same file, or when one branch deletes a file that the other branch modifies. Git cannot automatically determine which change to keep, requiring manual intervention.

COMMON CAUSES

- Simultaneous edits to the exact same lines of code in the same file by different developers on different branches.
- One branch deletes a file, while another branch modifies the same file.
- Changes in file permissions or renaming a file in one branch while it's modified in another.
- Modifications to binary files (e.g., images, compiled assets) that Git cannot intelligently merge.

A MINDSET FOR RESOLUTION

Rather than viewing merge conflicts as frustrating errors, consider them as **opportunities for code review and collaboration**. They are Git's way of preventing silent overwrites of important work, ensuring that all changes are intentionally integrated. Conflicts encourage developers to understand each other's changes and make informed decisions about the final codebase.

CONFLICT MARKERS EXPLAINED

When Git encounters a merge conflict, it inserts special markers into the conflicted file to highlight the sections that need manual resolution. Understanding these markers is the first step to resolving conflicts effectively.

```
<<<<<<< HEAD
Your version of the code here.
=======
Their version from the incoming branch.
>>>>>>> feature-branch
```

`<<<<<<< HEAD`

This marks the beginning of the conflicting changes from your current branch (`HEAD`). The code between `<<<<<<< HEAD` and `=======` is your version of the conflicting lines.

`=======`

This is the separator between the two conflicting versions. Everything above this line is from your branch, and everything below is from the incoming branch.

`>>>>>>> FEATURE-BRANCH`

This marks the end of the conflicting changes from the incoming branch (`feature-branch` in this example). The code between `=======` and this marker is their version of the conflicting lines.

HOW TO RESOLVE

To resolve a conflict, you must manually edit the file, deciding which version of the code to keep, or combining parts of both. Once you've made your decision, remove all conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`) and then use `git add <file-name>` to stage the resolved file.

For a more streamlined experience, consider using integrated development environment (IDE) tools like VS Code's built-in merge editor or Git's own `git mergetool`, which provide visual aids for conflict resolution.

RESOLVING A CONFLICT (STEP-BY-STEP)

When Git notifies you of a merge conflict, it's time to become the arbiter of your codebase. Follow these steps to systematically resolve the disagreement and finalize your merge.

01

IDENTIFY CONFLICTED FILES

Start by running `git status`. This command will clearly list all files that have unresolved conflicts, guiding you to where your attention is needed.

02

EDIT THE FILE

Open each conflicted file in your text editor. You'll find the conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`) outlining the conflicting sections. Decide which version of the code to keep, whether it's your changes, the incoming changes, or a combination of both.

03

REMOVE CONFLICT MARKERS

After deciding on the final content, delete all the conflict markers from the file. Ensure no `<<<<<<<`, `=======`, or `>>>>>>>` lines remain; your file should represent the desired code.

04

STAGE THE RESOLVED FILE

Once a file is clean of conflict markers and contains the correct code, use `git add <file-name>` to stage it. This tells Git you've finished resolving that particular file.

05

COMMIT THE MERGE

After all conflicted files are staged, execute `git commit`. Git will automatically provide a default merge message, which you can accept or modify to reflect the merge resolution. This completes the merge.

Should you find yourself overwhelmed or make a mistake during the resolution process, you can always abort the entire merge operation by running `git merge --abort`. This will revert your branch to its state before the merge attempt, allowing you to restart with a clean slate.

BEST PRACTICES TO AVOID & HANDLE CONFLICTS

Merge conflicts are an inevitable part of collaborative development, but proactive strategies and systematic resolution steps can minimize their impact and prevent significant delays.

FREQUENT SYNCHRONIZATION

Perform `git pull` regularly to fetch and integrate changes from the remote repository. Staying synced reduces the divergence between your local branch and the main codebase, making conflicts less likely.

SHORT-LIVED BRANCHES

Keep feature branches small and focused. Merge them back into the main branch frequently. The longer a branch lives, the more changes accumulate, increasing the probability and complexity of conflicts.

TEAM COMMUNICATION

Maintain open lines of communication within the team. Discuss planned changes, especially to shared files, using tools like Slack or issue trackers. Awareness of others' work can prevent conflicts before they even arise.

LINTER ENFORCEMENT

Use code linters and formatters (e.g., Prettier, ESLint) to ensure consistent code styling. Discrepancies in formatting can trigger unnecessary conflicts, even if the actual code logic hasn't changed.

By adopting these practices, teams can significantly streamline their Git workflow and minimize the disruption caused by merge conflicts, leading to more efficient development cycles.

BRANCHING IN REAL DEVOPS TEAMS

In a DevOps environment, efficient branching strategies and robust integration practices are critical for rapid, reliable software delivery. These workflows streamline development, testing, and deployment, minimizing friction and maximizing collaboration.

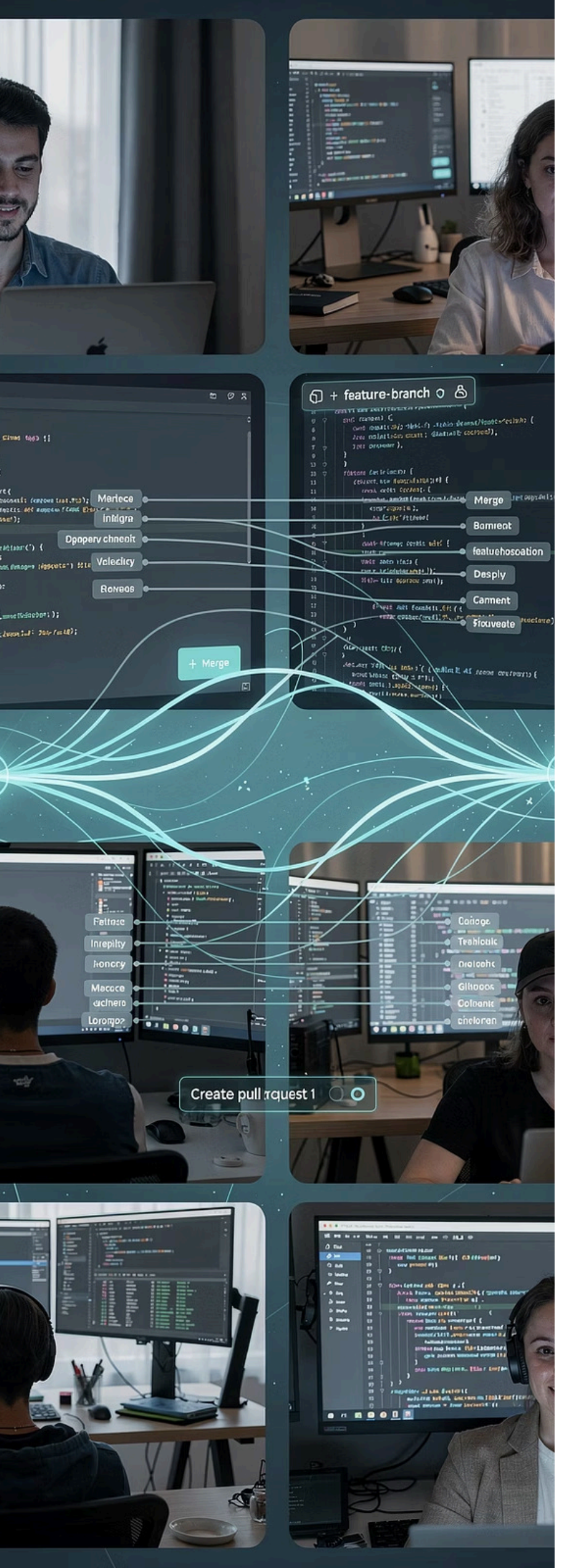
COMMON BRANCHING MODELS

- **GitFlow:** A strict, release-oriented model with dedicated branches for features, development, releases, and hotfixes.
- **GitHub Flow:** A simpler, continuous delivery approach where features are developed on branches, merged into `main` via pull requests, and deployed immediately.

INTEGRATION & AUTOMATION

- **Protected Branches:** Enforce rules like mandatory code reviews, status checks, and approval processes before merges are allowed into critical branches.
- **CI/CD Integration:** Automated builds, tests, and deployments are triggered by every merge, ensuring code quality and rapid feedback loops.

Prominent open-source projects, such as the Linux kernel, employ sophisticated branching hierarchies to manage contributions from thousands of developers worldwide, showcasing the power and flexibility of Git in large-scale collaborative efforts.



REMOTE BRANCHES & PULL REQUESTS

Remote branches and pull requests are the backbone of collaborative development in distributed teams, enabling efficient code sharing, review, and integration.

REMOTE SYNCHRONIZATION COMMANDS

Understanding these commands is crucial for keeping your local repository in sync with the remote without immediately altering your local work.

- `git fetch`: Downloads commits, files, and refs from a remote repository into your local repo. It's like checking what's new on the server without touching your current branch.
- `git pull origin main`: Fetches changes from the remote `main` branch and immediately merges them into your current local branch. This is a shorthand for `git fetch` followed by `git merge`.

THE POWER OF PULL REQUESTS (PRS)

Pull Requests (or Merge Requests on GitLab) are fundamental for integrating changes from a feature branch into a main branch.

- **Create on GitHub/GitLab:** Open a PR to propose changes and compare your branch with the target branch.
- **Discuss & Review:** Teammates can review code, suggest improvements, and engage in discussions directly on the platform.
- **Merge:** Once approved, your changes are merged into the main branch, often with automated checks.

Tip: Utilize **squash merges** to combine all commits from a feature branch into a single, clean commit upon merging, maintaining a tidy commit history.

COMMON PITFALLS & TROUBLESHOOTING

Even seasoned developers encounter challenges in Git. Understanding common issues and knowing how to effectively troubleshoot them can save valuable time and prevent potential data loss, keeping your workflow smooth and efficient.

ORPHANED BRANCHES

Sometimes local branches are no longer needed or linked to remote. Regularly clean up unused local branches with `git branch -d <branch-name>` after they've been merged to avoid clutter and confusion.

EXCESSIVE MERGE COMMITS

A history full of merge commits can be noisy. Strategically use `git rebase` to integrate changes cleanly, especially for feature branches that haven't been shared, creating a linear history.

LOST CHANGES

Accidentally deleting commits or branches is recoverable. Use `git reflog` to see a historical record of changes to your repository's HEAD, allowing you to restore lost work.

PERMISSION ERRORS

When pushing to a remote repository, you might encounter permission issues. Verify your SSH keys or HTTPS credentials, and ensure you have the necessary access rights configured on the remote hosting service.

Proactive learning and regular practice with Git commands are key to mastering these scenarios. Don't be afraid to experiment in a safe environment or consult the official Git documentation for complex situations.

EXERCISE DEBRIEF: REFLECTING ON GIT BRANCHING & CONFLICTS

Now that we've explored the core concepts and practices of Git branching, merging, and conflict resolution, let's take a moment to reflect on your experience and share insights.

WHAT SURPRISED YOU ABOUT BRANCHING?

Think about any aspects of creating, switching, or merging branches that challenged your expectations or offered new insights.

HOW DID RESOLVING THE CONFLICT FEEL?

Describe your experience with merge conflicts. Was it intuitive, frustrating, or empowering? What did you learn?

IDEAS FOR USING THIS IN YOUR PROJECTS?

Consider how you can apply these branching and merging strategies to your current or future development workflows.

Sharing your observations helps reinforce learning and builds collective understanding within the team.

RECAP & KEY TAKEAWAYS

We've covered a lot of ground in Git branching, merging, and conflict resolution. Here's a quick recap of the essential concepts and practices to carry forward into your projects.

BRANCHES ARE SAFE HAVENS

Use branches to isolate your work, experiment freely, and develop new features without impacting the main codebase until ready.

MERGE & REBASE STRATEGICALLY

Choose between merging for a clear history of feature integration or rebasing for a cleaner, linear project history, especially for local branches.

CONFLICTS ARE OPPORTUNITIES

Merge conflicts are a normal part of collaborative development. Approach them systematically and resolve them carefully to integrate changes correctly.

COLLABORATE & COMMUNICATE

Effective team communication, staying synced with remotes, and leveraging tools like Pull Requests are crucial for a smooth workflow.

Pro Tip: Build confidence by practicing Git commands on personal repositories. The more you use it, the more intuitive it becomes!

QUESTIONS, RESOURCES & NEXT STEPS

YOUR QUESTIONS & INSIGHTS

As we wrap up our session, let's open the floor for any questions and explore valuable resources to continue your Git journey with confidence.

CONTINUE YOUR GIT JOURNEY

Here are some excellent resources to deepen your understanding and master Git:

LINGERING CONFUSION?

Do you have any remaining questions about specific Git commands, branching strategies, or conflict resolution techniques?

- **Git Documentation**: The official and most comprehensive guide to Git's commands and concepts.
- **Interactive Git Tutorial**: A fun, visual, and interactive way to learn Git branching concepts.
- **"Pro Git" Book**: An in-depth, free online book covering all aspects of Git, from basics to advanced topics.