COMP 3550

3.2 — GITLAB WORKFLOW: BRANCHING, MRS, REVIEWS

Week 3: Version Control & Testing Foundations

HOW WE USE GITLAB IN THIS CLASS

One Repo Per Team

- Each team has its own GitLab repository
- All team members have shared access and collaboration rights

No Direct Pushes to main

- All changes must go through Merge Requests (MRs)
- Helps ensure code is reviewed before it’s merged
- Encourages clean history and teamwork

Why This Matters:

- Promotes accountability
- Catches bugs early
- Mimics professional team workflows

GITLAB BRANCHING STRATEGY

- Many Options Exist (see required readings/the internet)
- ***Feature Branch Workflow***
  - Create a new branch for each feature or bugfix
  - When done, open a Merge Request (MR)
  - After review, merge into main (no direct commits!)
  - Use dev or staging branches for larger teams or more complex projects
    - (e.g., test changes in staging before main)
    - not necessary here but many teams do like to have them
- **Branch Naming Tip:**
  - Use the format: *issue-###-short-description*
  - It’s up to your team to decide on a standard, document it, and use it
- **Examples:**
  - *issue-42-login-bug*

MERGE REQUESTS (MRS)

- Why Use MRs?
  - **Code Review:** Teammates can catch bugs or suggest improvements
  - **Traceability:** Links code changes to specific issues or tasks
  - **Quality Control:** Ensures code is tested and approved before merging
  - **Team Knowledge Transfer:** Everyone on the team should be able to understand the code that is going into the project
- When creating an MR:
  - **Clear Description** of what the branch does
  - **Issue Reference** (e.g., Closes #42) to auto-link it
  - **Test Notes**: how to reproduce and verify the change

TIPS FOR REVIEWING AN MR IN GITLAB

1. **Start with a High-Level Overview**
    - Read the MR description, linked issues, labels, etc. to understand context

TIPS FOR REVIEWING AN MR IN GITLAB

1. **Start with a High-Level Overview**
   - Read the MR description, linked issues, labels, etc. to understand context
2. **Review the Diff Carefully**
   - Skim changed files first to get a sense of scope
   - Then read line-by-line, checking correctness, style, edge cases, and testing
   - Leave comments on specific lines using “Start a review” so feedback is grouped

TIPS FOR REVIEWING AN MR IN GITLAB

1. **Start with a High-Level Overview**
   - Read the MR description, linked issues, labels, etc. to understand context
2. **Review the Diff Carefully**
   - Skim changed files first to get a sense of scope
   - Then read line-by-line, checking correctness, style, edge cases, and testing
   - Leave comments on specific lines using “Start a review” so feedback is grouped
3. **Focus on Quality Aspects**
   - Does it follow project guidelines (e.g., style, performance, compatibility)?
   - Are edge cases addressed?
   - Have tests been added or updated? Are they passing? (You might need to pull down their branch to test this ie. Step 4)

TIPS FOR REVIEWING AN MR IN GITLAB

1. **Start with a High-Level Overview**
   - Read the MR description, linked issues, labels, etc. to understand context
2. **Review the Diff Carefully**
   - Skim changed files first to get a sense of scope
   - Then read line-by-line, checking correctness, style, edge cases, and testing
   - Leave comments on specific lines using “Start a review” so feedback is grouped
3. **Focus on Quality Aspects**
   - Does it follow project guidelines (e.g., style, performance, compatibility)?
   - Are edge cases addressed?
   - Have tests been added or updated? Are they passing? (You might need to pull down their branch to test this ie. Step 4)
4. **Run the Code Locally if Possible**
   - Checkout the MR branch, test functionality, manually verify changes

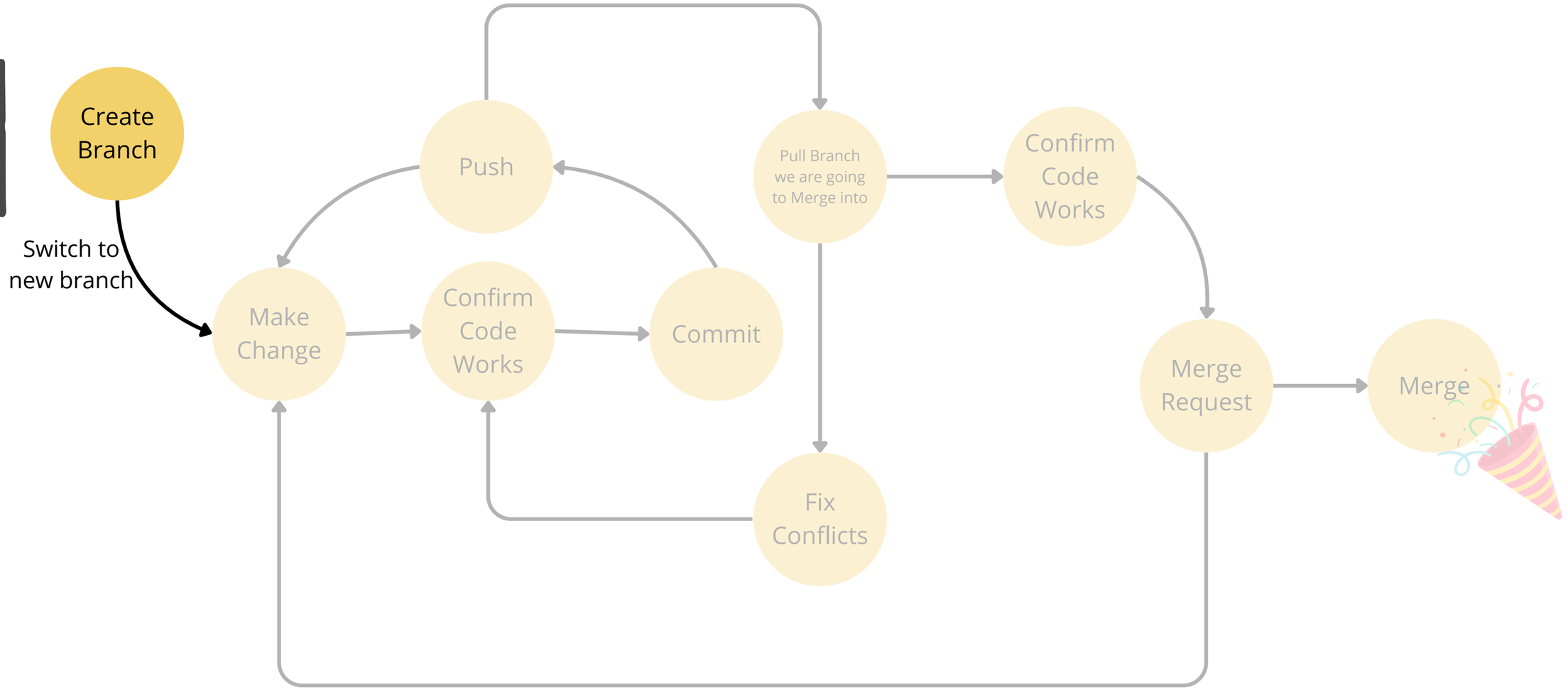
TIPS FOR REVIEWING AN MR IN GITLAB

1. **Start with a High-Level Overview**
   - Read the MR description, linked issues, labels, etc. to understand context
2. **Review the Diff Carefully**
   - Skim changed files first to get a sense of scope
   - Then read line-by-line, checking correctness, style, edge cases, and testing
   - Leave comments on specific lines using “Start a review” so feedback is grouped
3. **Focus on Quality Aspects**
   - Does it follow project guidelines (e.g., style, performance, compatibility)?
   - Are edge cases addressed?
   - Have tests been added or updated? Are they passing? (You might need to pull down their branch to test this ie. Step 4)
4. **Run the Code Locally if Possible**
   - Checkout the MR branch, test functionality, manually verify changes
5. **Provide Constructive, Kind Feedback**
   - Use /submit_review when done to bundle comments
   - Write actionable summaries: approve if ready, or request changes with clear next steps

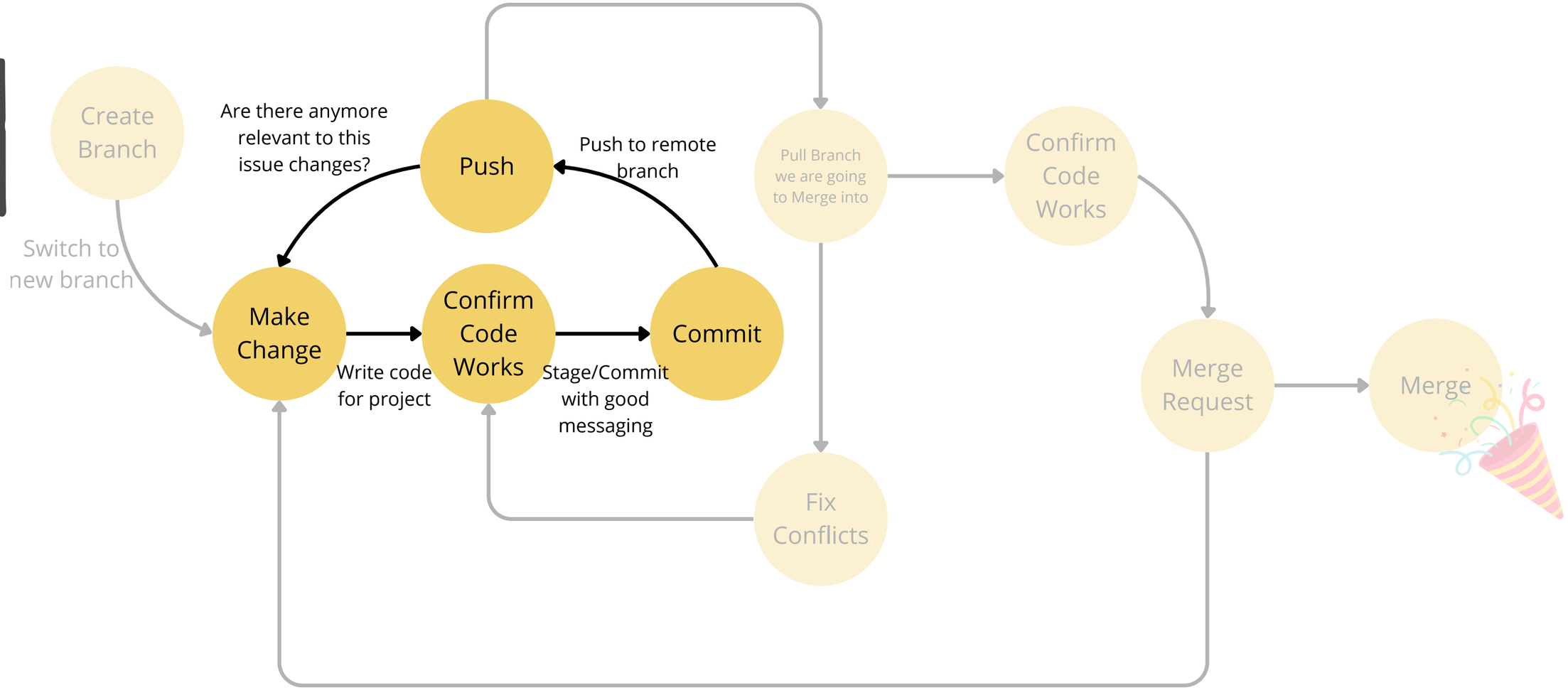
TIPS FOR REVIEWING AN MR IN GITLAB

1. **Start with a High-Level Overview**
   - Read the MR description, linked issues, labels, etc. to understand context
2. **Review the Diff Carefully**
   - Skim changed files first to get a sense of scope
   - Then read line-by-line, checking correctness, style, edge cases, and testing
   - Leave comments on specific lines using “Start a review” so feedback is grouped
3. **Focus on Quality Aspects**
   - Does it follow project guidelines (e.g., style, performance, compatibility)?
   - Are edge cases addressed?
   - Have tests been added or updated? Are they passing? (You might need to pull down their branch to test this ie. Step 4)
4. **Run the Code Locally if Possible**
   - Checkout the MR branch, test functionality, manually verify changes
5. **Provide Constructive, Kind Feedback**
   - Use /submit_review when done to bundle comments
   - Write actionable summaries: approve if ready, or request changes with clear next steps
6. **Post-Review Workflow**
   - If all clear, click Approve then Merge
   - Optionally delete source branch and update labels or milestones (as required)

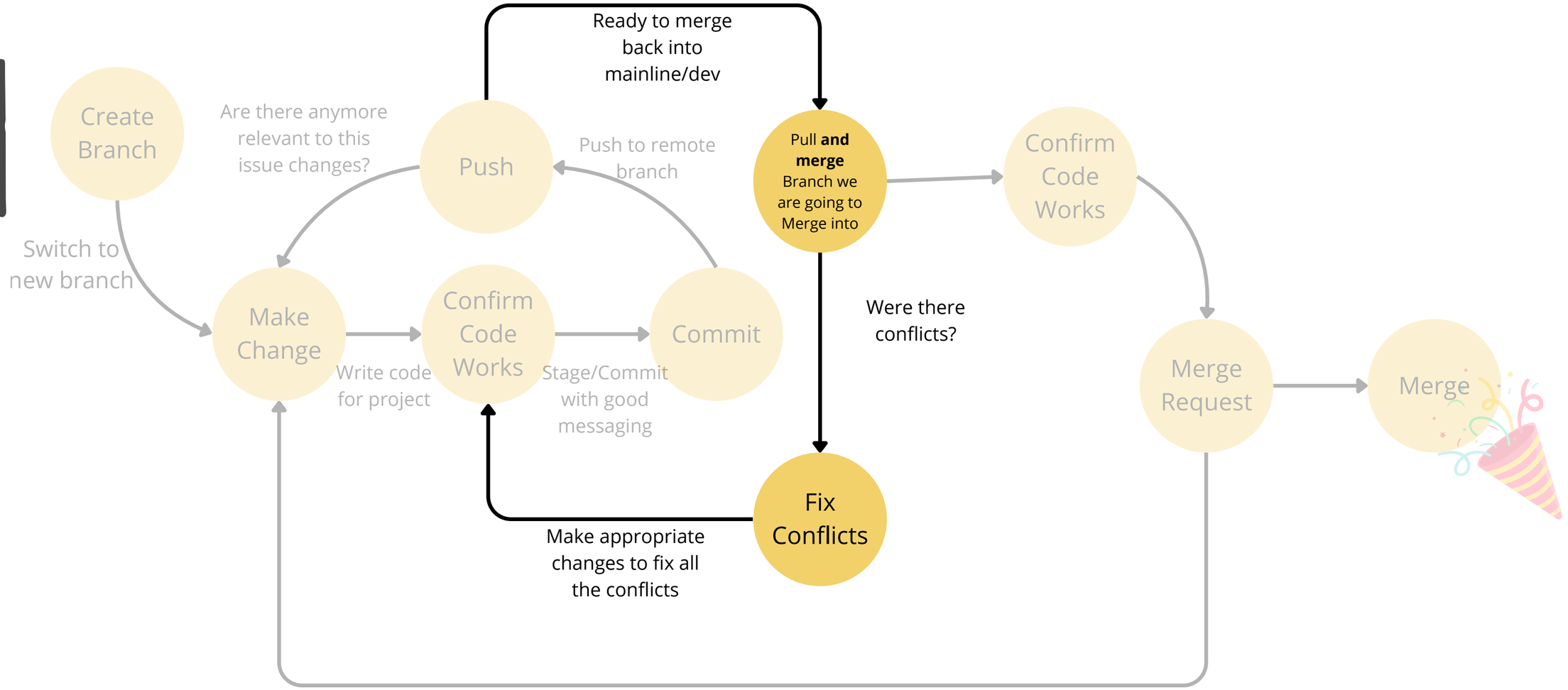
SAMPLE WORKFLOW



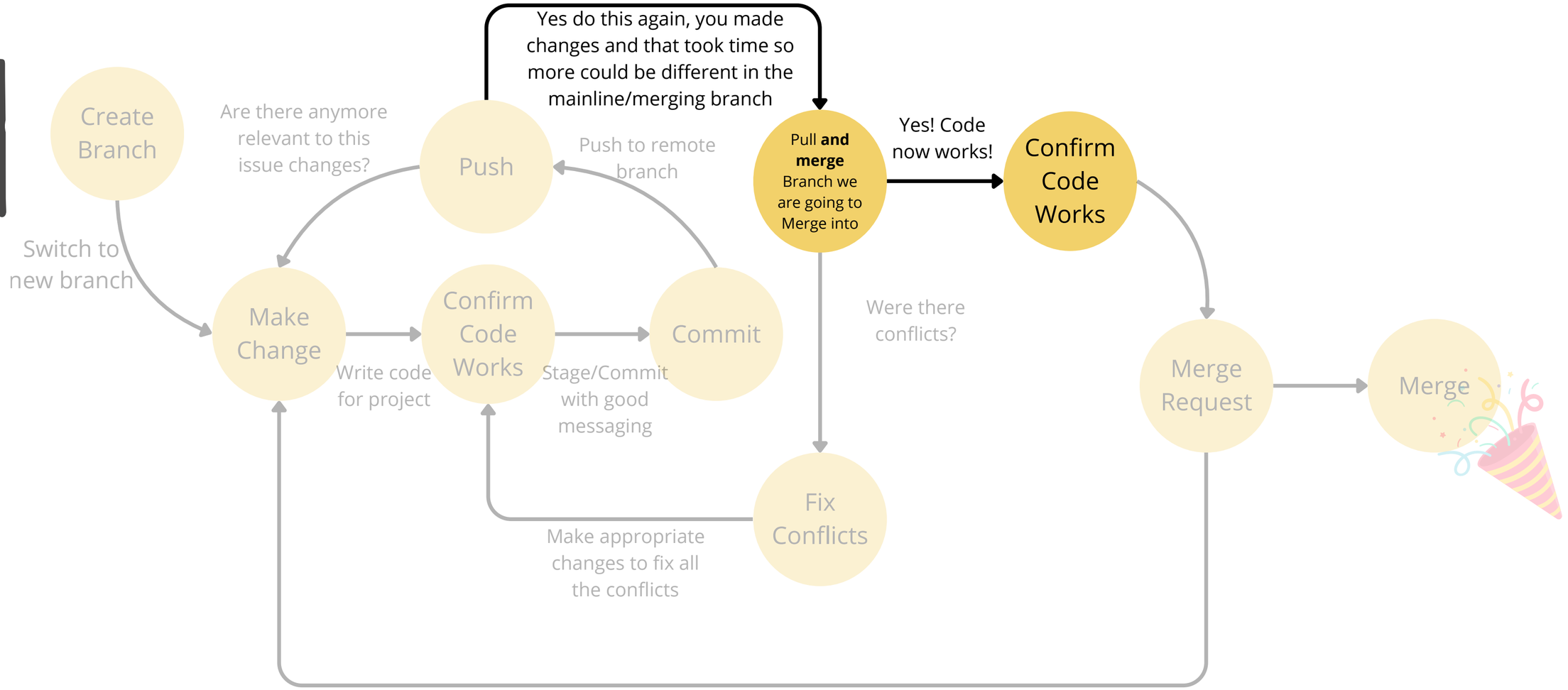
SAMPLE WORKFLOW



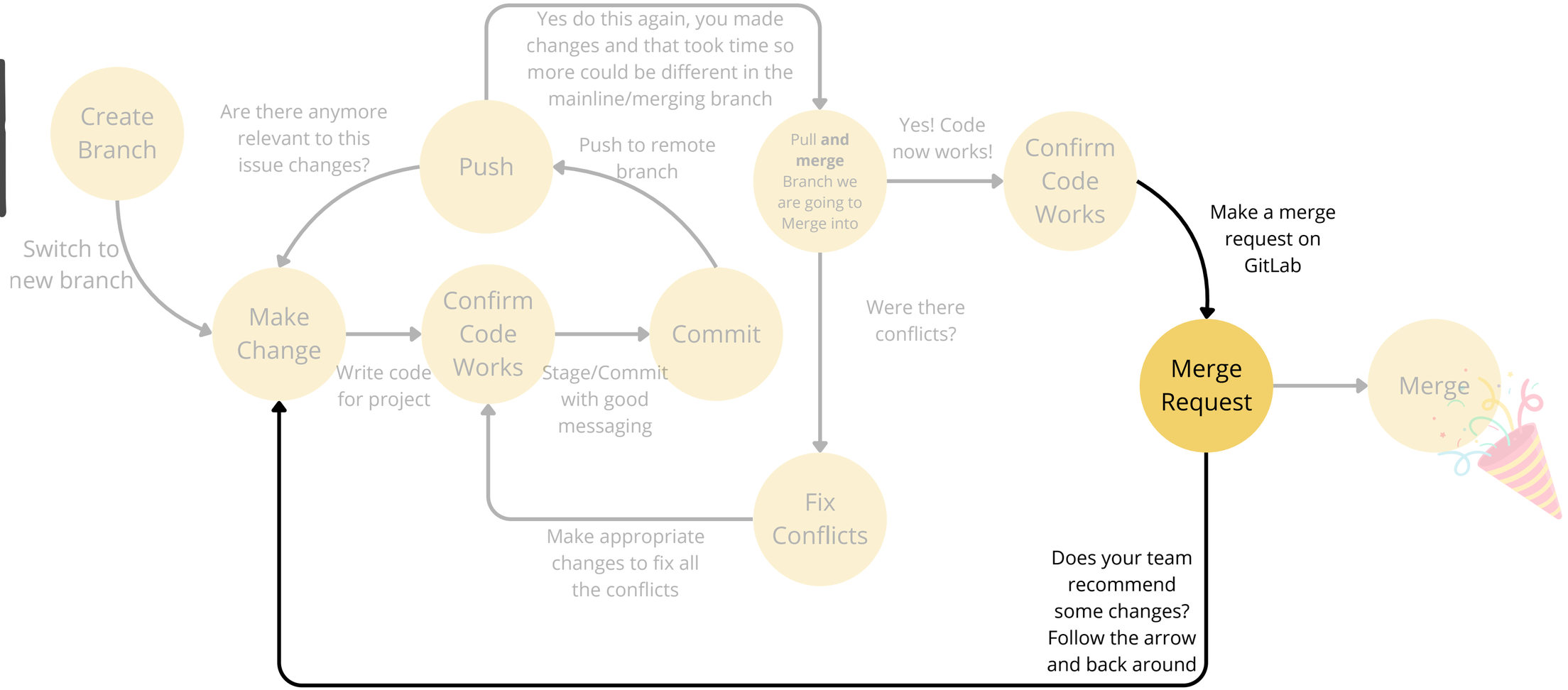
SAMPLE WORKFLOW



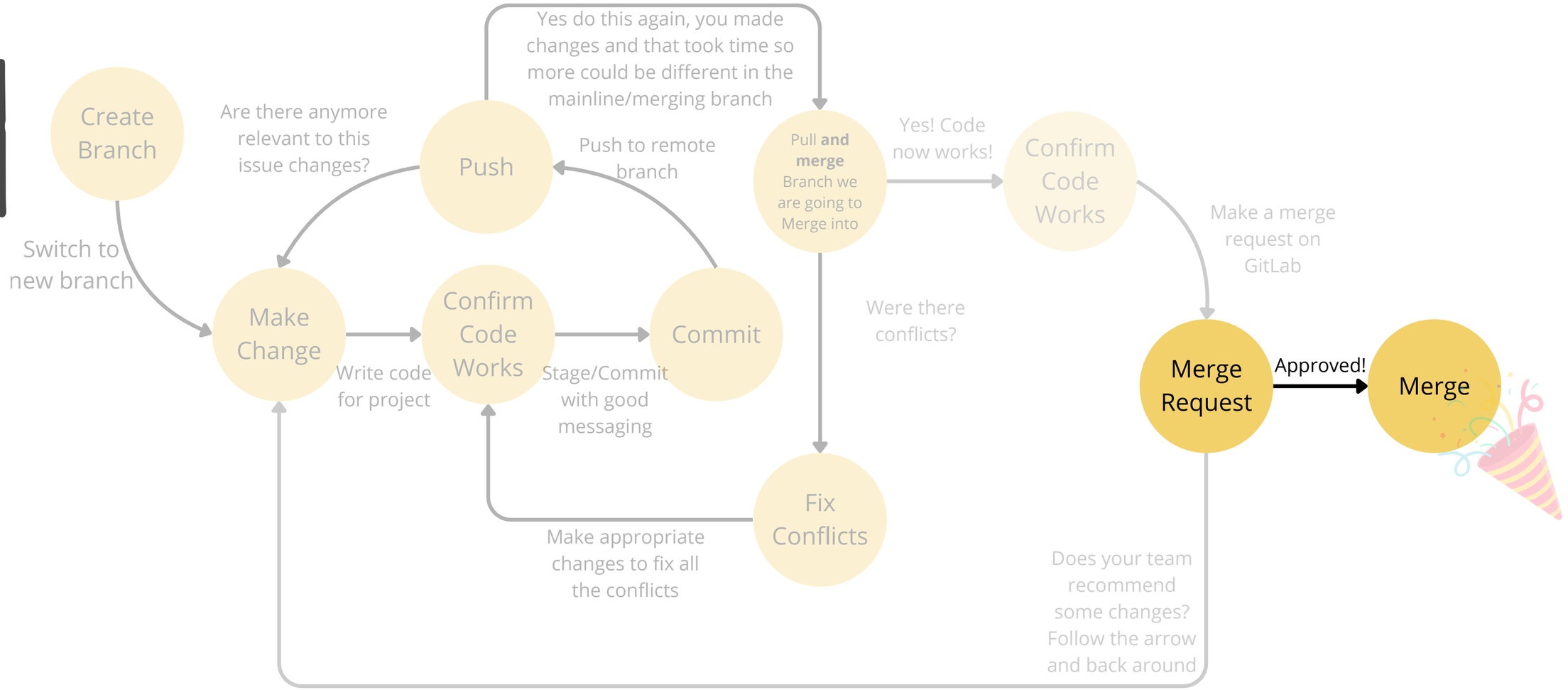
SAMPLE WORKFLOW



SAMPLE WORKFLOW



SAMPLE WORKFLOW

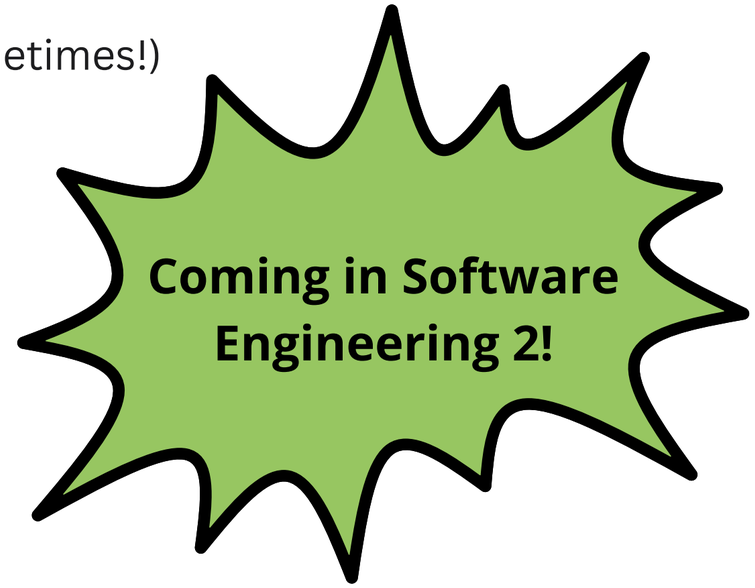


CONTINUOUS INTEGRATION (PREVIEW)

- **What It Is:**
  - CI = Continuous Integration
  - Automatically runs tests every time you push code to the repo
- On every push or Merge Request:
  - CI kicks off
  - Runs your automated tests
  - Flags problems before code is merged (or before MR sometimes!)
- What You'll See in GitLab:
  - Pass/fail status checks right on your MR
  - Click to view logs, errors, and test results
- Why It Matters:
  - Catches bugs early
  - Builds confidence in your code
  - Keeps main stable

CONTINUOUS INTEGRATION (PREVIEW)

- **What It Is:**
  - CI = Continuous Integration
  - Automatically runs tests every time you push code to the repo
- On every push or Merge Request:
  - CI kicks off
  - Runs your automated tests
  - Flags problems before code is merged (or before MR sometimes!)
- What You'll See in GitLab:
  - Pass/fail status checks right on your MR
  - Click to view logs, errors, and test results
- Why It Matters:
  - Catches bugs early
  - Builds confidence in your code
  - Keeps main stable



PAUSE & REFLECT

Take some time to explore the required readings and linked video resources on branching strategies.

Ask yourself:

- Which strategies seem practical or effective?
- Which ones don’t resonate with you — and why?
- How do these approaches align (or clash) with the way you like to work?

Use these reflections to start shaping a branching workflow that fits your team’s style and your own development habits.