Optimisation

1. **Clonage superficiel (`shallow clone`)**
   - **Sans**

     ```yaml
     - uses: actions/checkout@v4
       with:
         fetch-depth: 0    # récupère tout l'historique
     ```

   - **Avec**

     ```yaml
     - uses: actions/checkout@v4
       with:
         fetch-depth: 1    # ne récupère que le dernier commit
     ```

   - **Explication**

     Limiter `fetch-depth` à 1 réduit drastiquement le temps et la bande passante du `git clone` , ce qui accélère l'étape de checkout.
2. **Mise en cache des dépendances npm**
   - **Sans**

     ```yaml
     - run: npm ci
     ```

   - **Avec**

     ```yaml
     - uses: actions/cache@v4
       with:
         path: ~/.npm
         key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
     - run: npm ci
     ```

   - **Explication**

     En stockant `~/.npm` entre les runs, `npm ci` réutilise les paquets déjà téléchargés au lieu de tout retélécharger à chaque exécution.
3. **Installation optimisée**
   - **Sans**

     ```yaml
     - run: npm install
     ```

- **Avec**

```
- run: npm ci --prefer-offline --no-audit
```

- **Explication**
  `npm ci` est plus rapide et plus fiable pour les environnements CI, et les flags `--prefer-offline` et `--no-audit` suppriment les vérifications réseau et audit, gagnant quelques secondes.

4. **Regrouper plusieurs commandes en un seul step**
   - **Sans**

```
- run: npm ci
- run: npm run lint
- run: npm test
```

   - **Avec**

```
- run: |
    npm ci
    npm run lint
    npm test
```

   - **Explication**
     Chaque `run` lance un nouveau shell: regrouper réduit le nombre de spawns et accélère légèrement l'exécution globale.
5. **Utiliser un conteneur pré-configuré**
   - **Sans**

```
runs-on: ubuntu-latest
steps:
  - uses: actions/checkout@v4
  - run: npm ci && npm test
```

   - **Avec**

```
runs-on: ubuntu-latest
container:
  image: node:18-alpine
steps:
  - uses: actions/checkout@v4
  - run: npm ci && npm test
```

- **Explication**
  Le conteneur `node:18-alpine` embarque Node.js et npm, évitant d'installer des dépendances système à chaque run et réduisant le temps de setup.

6. **Limiter l'exécution aux fichiers modifiés**
   - **Sans**

     ```yaml
     on: [push, pull_request]
     ```

   - **Avec**

     ```yaml
     on:
       push:
         paths:
           - 'src/**'
           - 'package.json'
     ```

   - **Explication**
     Le job ne se déclenche que si des fichiers pertinents pour Node.js changent, économisant des minutes de build sur d'autres types de modifications.
7. **Annulation automatique des runs en attente (`concurrency`)**
   - **Sans**

     ```yaml
     jobs:
       build:
         runs-on: ubuntu-latest
     ```

   - **Avec**

     ```yaml
     concurrency:
       group: ${{ github.ref }}
       cancel-in-progress: true
     jobs:
       build:
         runs-on: ubuntu-latest
     ```

   - **Explication**
     Quand un nouveau push arrive, GitHub annule les runs en cours pour la même branche, réduisant la file d'attente et évitant des builds redondants.
8. **Utiliser un runner self-hosted pour plus de CPU/mémoire**
   - **Sans**

```
runs-on: ubuntu-latest
```

- **Avec**

```
runs-on: [self-hosted, linux, x64]
```

- **Explication**
  Les self-hosted runners restent chauds et peuvent avoir plus de ressources (par ex. 8 cœurs, 32 Go RAM), éliminant le temps de provisionning et accélérant les builds lourds.

9. **Exécution parallèle via matrix (variantes Node.js)**
   - **Sans**

```
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - run: npm ci && npm test
```

   - **Avec**

```
jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [16, 18, 20]
    steps:
      - uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node-version }}
      - run: npm ci && npm test
```

   - **Explication**
     Tester en parallèle sur plusieurs versions de Node.js permet de valider la compatibilité plus rapidement qu'en séquentiel.

10. **Mettre en cache les modules Node globalement**
    - **Sans**

```
- run: npx some-tool
```

    - **Avec**

```
- uses: actions/cache@v4
  with:
    path: ~/.npm-global
    key: ${{ runner.os }}-global-${{ hashFiles('**/package.json') }}
- run: |
    npm config set prefix ~/.npm-global
    npm install -g some-tool
    some-tool
```

- **Explication**
  Mettre en cache `~/.npm-global` permet de réutiliser les outils CLI installés globalement, évitant les installations répétitives.

11. **Passer à un gestionnaire de paquets ultra-rapide**
    - **Sans**

      ```
      - run: npm ci
      ```

    - **Avec** (pnpm)

      ```
      - uses: pnpm/action-setup@v2
        with:
          version: 8
      - run: pnpm install --frozen-lockfile
      ```

    - **Explication**
      `pnpm` déduplique les dépendances dans un store global et installe en parallèle, réduisant souvent le temps d'installation de 30–50 % par rapport à `npm`.
12. **Utiliser la restauration de cache progressive (`restore-keys`)**
    - **Sans**

      ```
      - uses: actions/cache@v4
        with:
          path: ~/.npm
          key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
      ```

    - **Avec** (fallback partiel)

```yaml
- uses: actions/cache@v4
  with:
    path: ~/.npm
    key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
    restore-keys: |
      ${{ runner.os }}-node-
      ${{ runner.os }}-
```

- **Explication**
  Si la clé exacte échoue (nouveau `package-lock.json`), GitHub restaure un cache plus ancien et partiel, accélérant quand même l'installation au lieu de repartir de zéro.

13. **Exécuter uniquement les tests impactés (`jest --changedSince`)**
    - **Sans**

      ```yaml
      - run: npm test
      ```

    - **Avec**

      ```yaml
      - run: |
          npx jest --changedSince=${{ github.event.before }}
      ```

    - **Explication**
      Jest ne relance que les tests des fichiers modifiés depuis le dernier commit, ce qui peut réduire drastiquement le temps de test sur de gros projets.
14. **Cache local de build incrémental avec Turborepo/Nx**
    - **Sans**

      ```yaml
      - run: npm run build
      ```

    - **Avec** (Turborepo)

      ```yaml
      - uses: actions/cache@v4
        with:
          path: .turbo
          key: ${{ runner.os }}-turbo-${{ hashFiles('**/turbo.json') }}
      - run: npx turbo run build
      ```

    - **Explication**
      Turborepo conserve un cache local des artefacts de build (`.turbo`), évitant de rebuild des paquets non modifiés, et se combine à GitHub Actions pour persister entre les runs.
15. **Docker BuildKit avec export/import de cache**

- **Sans**

```
- run: docker build -t app:latest .
```

- **Avec**

```
- run: |
    docker buildx create --use
    docker buildx build \
      --cache-from=type=gha \
      --cache-to=type=gha,mode=max \
      --tag app:latest .
```

- **Explication**
  BuildKit synchronise le cache des couches Docker via l'intégration GitHub Actions (`type=gha`), ce qui accélère les rebuilds en réutilisant les couches inchangées.

16. **Partager des artefacts entre jobs**
    - **Sans**

```
# chaque job doit rebuild
```

    - **Avec**

```
jobs:
  build:
    steps:
      - run: npm run build
      - uses: actions/upload-artifact@v3
        with:
          name: build-output
          path: dist/
  test:
    needs: build
    steps:
      - uses: actions/download-artifact@v3
        with:
          name: build-output
          path: dist/
      - run: npm test -- --testPathPattern dist/
```

    - **Explication**
      En uploadant le dossier `dist/` une seule fois, les autres jobs (tests, lint, packaging)

peuvent réutiliser les artefacts déjà générés, supprimant les rebuilds redondants.

17. **Préchauffage de runners self-hosted (Warm-up)**
    - **Sans**

      ```
      # runner idle → provisioning à chaque run
      ```

    - **Avec** (script d'auto-wake)

      ```yaml
      on:
        schedule:
          - cron: '*/5 * * * *'
      jobs:
        wake-up:
          runs-on: [self-hosted]
          steps:
            - run: echo "Keeping runner warm"
      ```

    - **Explication**
      Un job planifié toutes les 5 minutes empêche un self-hosted runner de passer en idle, garantissant qu'il reste prêt pour vos jobs principaux sans latence de provisionning.
18. **Utiliser des runners « éphémères » via HashiCorp Waypoint / Kubernetes**
    - **Sans**

      ```yaml
      runs-on: self-hosted
      ```

    - **Avec** (K8s Runner Controller)

      ```yaml
      runs-on: [kubernetes, linux]
      ```

    - **Explication**
      Les runners éphémères sur Kubernetes (avec Actions Runner Controller) sont lancés à la demande, puis détruits après exécution, assurant un environnement propre et évitant la dérive over time.
19. **Contrôler les étapes conditionnelles**
    - **Sans**

      ```yaml
      - run: npm run deploy
      ```

    - **Avec**

```
- run: npm run deploy
  if: github.event_name == 'workflow_dispatch' && github.ref == 'refs/heads/mai
```

- **Explication**
  L'étape (ou le job) ne s'exécute que si elle est pertinente (par ex. déploiement manuel sur `main`), évitant des opérations inutiles sur d'autres branches.

20. **Prise en charge des monorepos via `paths-ignore`**
    - **Sans**

      ```
      on: push
      ```

    - **Avec**

      ```
      on:
        push:
          paths-ignore:
            - 'docs/**'
            - 'README.md'
      ```

    - **Explication**
      Dans un monorepo, ignorez les modifications sur les dossiers non liés au code Node (docs, config) pour ne lancer les workflows que lorsque le code métier change.