ChatGPT

Step 4 – Progettazione e miglioramento della pipeline CI/CD

Analisi dello stato attuale della pipeline

Le automazioni CI/CD già configurate nel repository Evo-Tactics sono piuttosto articolate e coprono molti aspetti del ciclo di sviluppo. Fra i workflow principali presenti nella directory `.github/workflows` si distinguono:

Workflow	Finalità principali	Note rilevanti
CI (`ci.yml`)	È il fulcro della CI. Usa un job `paths-filter` per stabilire quali parti del repo sono cambiate e lancia in modo condizionale una serie di job: build del bundle Playwright per gli smoke test del deploy, test TypeScript (`npm test` e validazione specie TS), qualità webapp (lint, test, build e preview), check CLI e dataset (compilazione, validazione dataset, run di `cli_smoke.sh`), test Python (`pytest`, validazione specie, roll pack), controlli dei dataset (audit e rigenerazione baseline e report di copertura dei tratti, generazione dashboard, controlli di stile, refresh report di stato) e verifica della conformità allo style guide [1].	
Data Quality (`data-quality.yml`)	Esegue audit sui tratti e validazione degli YAML quando una PR tocca dati o script correlati. Installa dipendenze Node e Python, esegue `validate_datasets.py`, lancia l'audit dei tratti in modalità check, genera l'indice dei tratti, produce un report di copertura severo e la dashboard di completamento, quindi carica i report generati [2].	
QA Reports (`qa-reports.yml`)	Genera baseline QA (badge, baseline dei tratti) tramite `scripts/export-qa-report.js`, controlla che i report siano aggiornati; se differiscono dal repo, il job segnala di eseguire `npm run export:qa` [3].	
HUD Canary (`hud.yml`)	Abilita/disabilita la build dell'overlay HUD in base al flag `default:` presente in `config/cli/hud.yaml`. Se il flag è attivo, imposta Node, esegue `npm ci` e compila l'overlay; altrimenti salta la build [4].	
Validate registry naming (`validate-naming.yml`)	Controlla consistenza dei nomi nel registro delle specie e dei tratti eseguendo `validate_registry_naming.py` sui file interessati [5].	

Questi workflow mostrano una pipeline complessa ma modulare: per ogni area (TS, webapp, CLI, Python, dataset, style guide, deploy) ci sono job dedicati con set-up di Node e Python. L'uso di `actions/cache` è limitato alle dipendenze npm; non ci sono controlli di sicurezza, né fasi di scanning o threat modeling.

Best practice di riferimento

La letteratura suggerisce di strutturare una pipeline CI/CD in fasi ben definite (origine, build, test, distribuzione) e di integrare in ciascuna fase controlli di qualità e di sicurezza [6]. La metodologia DevOps raccomanda commit frequenti, test automatizzati e monitoraggio continuo [7]. Inoltre il modello Secure SDLC prevede l'integrazione di analisi dei rischi, threat modeling e test di sicurezza in tutte le fasi del ciclo di vita [8].

Raccomandazioni e possibili miglioramenti

1. **Riorganizzare i job per fasi logiche**
   Il file `ci.yml` contiene numerosi job che installano ripetutamente Node e Python. Si può semplificare seguendo le quattro fasi canoniche del CI/CD [6]:
2. *Origine*: mantenere il job `paths-filter` per determinare l'area impattata e generare output booleani.
3. *Build*: un job unico per compilare i componenti TS (CLI e webapp) e Python se necessario, usando `npm run build` e `python setup.py` / `pip install -e` con caching centralizzato.
4. *Test*: raggruppare i test unitari TS e Python e la validazione dei dataset in due job distinti (frontend/backend/dataset). Usare matrice di esecuzione per versioni Node/Python diverse solo se necessario.

5. *Deploy*: mantenere il job `deployment-checks` con i passaggi di validazione e smoke test, ma prevedere step condizionati per release taggate.

6. **Caché ottimizzati e reuse degli ambienti**
   Attualmente ogni job reinstalla tutte le dipendenze. L'uso di `actions/cache` per i pacchetti Python può velocizzare l'esecuzione (cache di `~/.cache/pip`) e le dipendenze Node (cache di `~/.npm`). Si può anche sfruttare la funzione `install-dependencies` in un job di base e passare gli artefatti agli altri job tramite `needs` e `outputs`.

7. **Integrare scansioni di sicurezza e analisi statica/dinamica**

8. **SAST**: eseguire `bandit` o `pylint` per il codice Python e `npm audit` / `eslint` per i moduli TS.
9. **Gestione segreti**: aggiungere uno step con `truffleHog` o `git-secrets` per rilevare credenziali accidentali.
10. **Threat modeling**: inserire un documento di threat model (parte del GDD) e definire checklists in uno script da eseguire in CI per valutare l'impatto di nuove funzionalità [8].

11. **Code QL**: considerare l'attivazione dell'analisi Code QL integrata di GitHub per vulnerabilità note.

12. **Quality gates e metriche di copertura**

13. Impostare soglie minime di copertura dei test e di completamento dei tratti nel dataset. In `ci.yml` esistono già script che falliscono se mancano specie o se la copertura dei tratti è sotto

soglia [9] ; questi check vanno mantenuti ma integrati con soglie per la copertura dei test TS/Python.

14. Esportare i report di test e copertura come artefatti e integrarli in dashboards (es. con `codecov` o badge nel README).

15. **Unificare la validazione YAML e l'audit dei tratti**
    I workflow `data-quality.yml` e `ci.yml` eseguono controlli simili (audit, index e copertura). Consolidare questi passaggi in un'unica job "Dataset checks" riduce duplicazioni. Si può parametrizzare la severità (check vs. strict) tramite input GitHub Actions.

16. **Controlli di naming e style guide**
    `validate-naming.yml` e `styleguide-compliance` sono separati; unificare i controlli di stile in un job "Static analysis & style" che esegue naming, lint, formattazione e generazione report. Caricare gli output come artefatti e pubblicare un badge di conformità.

17. **Incrementalizzazione e concurrency**

18. Utilizzare `concurrency` e `workflow_dispatch` per impedire l'avvio di più pipeline contemporanee sulla stessa branch.

19. Per i dataset, attivare la pipeline solo quando i file in `data/` o `packs/` cambiano, riducendo l'esecuzione non necessaria.

20. **Monitoraggio e feedback rapido**
    Inserire step che commentano automaticamente nelle pull request i risultati di audit e test, fornendo link agli artefatti. Utilizzare GitHub Projects o Slack webhooks per notificare failure in tempo reale.

Prossimi passi

- Redigere un documento interno (es. `docs/devops-ci-cd.md`) che descriva la pipeline riprogettata, le responsabilità di ciascun job e come aggiungere nuovi check.
- Aggiornare i workflow GitHub secondo le raccomandazioni sopra e testare la pipeline in ambiente di staging.
- Integrare la cultura DevOps con formazione al team su CICD e secure coding, come suggerito dalle best practice [7] .

Conclusioni

La pipeline attuale copre molti aspetti (build, test, dataset, HUD, naming), ma può essere semplificata e resa più robusta seguendo le quattro fasi standard e integrando controlli di sicurezza e di qualità più completi. Una riorganizzazione modulare, unita a caching e gating centralizzati, ridurrà i tempi di esecuzione e aumenterà l'affidabilità del rilascio.

---

19 ci.yml
https://github.com/MasterDD-L34D/Game/blob/HEAD/.github/workflows/ci.yml

2 data-quality.yml
https://github.com/MasterDD-L34D/Game/blob/HEAD/.github/workflows/data-quality.yml

3 qa-reports.yml
https://github.com/MasterDD-L34D/Game/blob/HEAD/.github/workflows/qa-reports.yml

4 hud.yml
https://github.com/MasterDD-L34D/Game/blob/HEAD/.github/workflows/hud.yml

5 validate-naming.yml
https://github.com/MasterDD-L34D/Game/blob/HEAD/.github/workflows/validate-naming.yml

6 Che cos'è la pipeline CI/CD? Integrazione e distribuzione continue | Fortinet
https://www.fortinet.com/it/resources/cyberglossary/ci-cd-pipeline

7 Pipeline DevOps | Atlassian
https://www.atlassian.com/it/devops/devops-tools/devops-pipeline

8 (Microsoft Word - Allegato 1- Linee Guida per l\222adozione di un ciclo di sviluppo di software sicuro)
https://www.agid.gov.it/sites/agid/files/2024-06/Linee_guida_per_ladozione_di_un_ciclo_di_sviluppo_di_software_sicuro.pdf