वितरित Git (Distributed Git)

जब तपाईंले एक **रिमोट Git भण्डारण (Remote Git Repository)** सेटअप गरिसक्नुभयो, जसले सबै विकासकर्ताहरूलाई आफ्नो कोड साझा गर्न सुविधा प्रदान गर्छ, त्यसपछि तपाईंले Git को वितरित कार्यप्रवाह (Distributed Workflow) कसरी उपयोग गर्ने भन्ने सिक्नुपर्छ।

यस अध्यायमा, तपाईंले Git लाई **वितरित वातावरण (Distributed Environment)** मा कसरी प्रयोग गर्ने भन्ने सिक्नुहुनेछ, दुबै **योगदानकर्ता (Contributor)** र **समावेशकर्ता (Integrator)** को रूपमा।

**योगदानकर्ता (Contributor) को रूपमा काम गर्ने तरिका**

कुनै पनि परियोजनामा सफलतापूर्वक योगदान गर्न, तपाईंले निम्न चरणहरू अनुसरण गर्न सक्नुहुन्छ:

1. **परियोजना क्लोन गर्नुहोस् (Clone the Project):**
   * पहिले, Git भण्डारणलाई आफ्नो स्थानीय प्रणालीमा डाउनलोड गर्नुहोस्।

git clone https://server/namespace/project.git

यसले तपाईंलाई परियोजनाको सम्पूर्ण इतिहास प्राप्त गर्न अनुमति दिन्छ।

1. **नयाँ शाखा सिर्जना गर्नुहोस् (Create a New Branch):**
   * परिवर्तनहरू नगर्ने बेला कहिल्यै main (वा master) शाखामा प्रत्यक्ष कार्य नगर्नुहोस्।

git checkout -b feature-branch

* + यसले तपाईँको नयाँ विशेषता वा सुधारहरूका लागि छुट्टै शाखा सिर्जना गर्छ।

1. **परिवर्तनहरू लागू गर्नुहोस् (Make Changes & Commit):**
   * आवश्यक परिवर्तनहरू गर्नुहोस् र तिनलाई स्थानीय रूपमा प्रतिबद्ध गर्नुहोस्।

git add .

git commit -m "Add new feature"

1. **रिमोट भण्डारणमा पुश गर्नुहोस् (Push to Remote Repository):**
   * तपाईंको शाखालाई रिमोट GitLab भण्डारणमा पठाउनुहोस्।

git push origin feature-branch

1. **मर्ज अनुरोध (Merge Request) खोल्नुहोस्:**
   * GitLab को वेब UI प्रयोग गरेर, आफ्नो शाखाबाट main शाखामा मर्ज अनुरोध पठाउनुहोस्।
   * परियोजना प्रशासकहरूले समीक्षापछि तपाईंको परिवर्तनहरू समावेश गर्नेछन्।

**समावेशकर्ता (Integrator) को रूपमा काम गर्ने तरिका**

परियोजना प्रशासक वा समावेशकर्ताको रूपमा तपाईंले योगदानकर्ताहरूको परिवर्तनहरू व्यवस्थापन गर्नुपर्नेछ। सफल परियोजना सञ्चालनका लागि निम्न कार्यहरू गर्नुहोस्:

1. **योगदान समीक्षा गर्नुहोस् (Review Contributions):**
   * योगदानकर्ताले पठाएका मर्ज अनुरोधहरू समीक्षा गर्नुहोस्।
   * कोड समीक्षा गर्दा ध्यान दिनुपर्ने बुँदाहरू:
     + कोड शैली नीति (Code Style Guidelines)
     + प्रदर्शन र सुरक्षा (Performance & Security)
     + एकीकरण परीक्षणहरू (Integration Tests)
2. **स्थानीय रूपमा परीक्षण गर्नुहोस् (Test Locally):**
   * मर्ज अनुरोध स्वीकृत गर्नु अघि स्थानीय रूपमा कोड परीक्षण गर्नुहोस्।

git fetch origin feature-branch

git checkout feature-branch

1. **मर्ज अनुरोधलाई समावेश गर्नुहोस् (Merge the Request):**
   * समीक्षापछि, मर्ज अनुरोधलाई परियोजनामा समावेश गर्नुहोस्।

git merge feature-branch

1. **वितरित कार्यप्रवाह व्यवस्थापन (Manage Distributed Workflow):**
   * योगदानकर्ताहरूको समन्वय गर्नको लागि GitLab मा **माइलस्टोनहरू (Milestones), मुद्दाहरू (Issues)** र **ट्यागहरू (Tags)** प्रयोग गर्नुहोस्।
   * CI/CD एकीकरणको प्रयोग गरेर गुणस्तर सुनिश्चित गर्नुहोस्।

**वितरित कार्यप्रवाहका फाइदाहरू (Benefits of Distributed Workflow)**

* **समन्वय (Collaboration):** कुनै पनि स्थानबाट धेरै विकासकर्ताहरूले सजिलै काम गर्न सक्छन्।
* **कोड गुणस्तर (Code Quality):** मर्ज अनुरोध प्रणालीले विकासकर्ताहरूलाई समीक्षा प्रक्रिया अपनाउन प्रेरित गर्छ।
* **ब्याकअप र सुरक्षाः** विभिन्न भण्डारणहरू बीच डाटा सुरक्षित राख्न सकिन्छ।
* **सहज परियोजना व्यवस्थापन:** GitLab को इन्टरफेसले प्रशासनिक कार्यहरू सजिलो बनाउँछ।

यसरी, वितरित Git कार्यप्रवाहमा योगदानकर्ता र समावेशकर्ताको रूपमा कार्य गरेर, परियोजनाको उत्पादकता, गुणवत्ता, र समन्वय सुधार गर्न सकिन्छ।

वितरित कार्यप्रवाहहरू (Distributed Workflows)

Git को **वितरित प्रकृति (Distributed Nature)** ले परियोजनामा विकासकर्ताहरूले बढी लचिलो रूपमा सहकार्य गर्न अनुमति दिन्छ। पारम्परिक **केन्द्रित संस्करण नियन्त्रण प्रणालीहरू (CVCSs)** जस्तै SVN वा Perforce मा, हरेक विकासकर्ता केन्द्रिय भण्डारण (Central Hub) सँग जडान हुन्छन्, जहाँ सम्पूर्ण परिवर्तनहरू सोही स्थानमा गरिन्छ।

तर, Git मा हरेक विकासकर्ता स्वयं **नोड (Node)** मात्र नभई **हब (Hub)** पनि हुन सक्छन्। यसले विकासकर्ताहरूलाई अन्य भण्डारणमा योगदान गर्न, आफ्नो भण्डारणहरू सार्वजनिक रूपमा उपलब्ध गराउन, र अन्य विकासकर्ताहरूले त्यसमा आधारित भएर कार्य गर्न अनुमति दिन्छ।

विकासकर्ताहरूलाई बढी स्वतन्त्रता दिने यस प्रणालीले विभिन्न प्रकारका **कार्यप्रवाह (Workflows)** अपनाउन सक्ने सुविधा दिन्छ। अब, हामी केही प्रमुख वितरित कार्यप्रवाहहरूको समीक्षा गर्नेछौं।

**1. केन्द्रित कार्यप्रवाह (Centralized Workflow)**

यस कार्यप्रवाहमा सबै विकासकर्ताहरूले **एकल रिमोट भण्डारण (Single Remote Repository)** प्रयोग गर्छन्, जुन केन्द्रिय सत्यको स्रोत (Single Source of Truth) को रूपमा कार्य गर्छ।

**प्रक्रिया:**

1. विकासकर्ताहरूले मुख्य भण्डारणबाट clone गर्छन्।
2. नयाँ शाखा सिर्जना गरेर परिवर्तनहरू गर्छन्।
3. आफ्नो परिवर्तनहरू push गर्छन् र मर्ज अनुरोध (Merge Request) पठाउँछन्।
4. मर्ज अनुरोध स्वीकृत भएपछि, परिवर्तनहरू main शाखामा समावेश गरिन्छ।

**फाइदाहरू:**

* सरल संरचना, नयाँ सदस्यहरूका लागि सजिलो।
* व्यवस्थापन र समीक्षा प्रक्रियालाई नियन्त्रण गर्न सजिलो।

**कमजोरीहरू:**

* स्केलेबिलिटी समस्या, किनकि सबैले एउटै भण्डारणमा निर्भर रहनुपर्छ।
* सानो टिमहरूका लागि उपयुक्त, तर ठूलो परियोजनामा बिस्तार गर्न गाह्रो।

**2. फीचर शाखा कार्यप्रवाह (Feature Branch Workflow)**

यस कार्यप्रवाहमा, प्रत्येक नयाँ सुविधा वा बग फिक्सलाई छुट्टै शाखामा विकास गरिन्छ, जुन मुख्य शाखामा समावेश गर्नु अघि समीक्षा गरिन्छ।

**प्रक्रिया:**

1. main शाखाबाट छुट्टै शाखा (feature-xyz) सिर्जना गर्नुहोस्।
2. आवश्यक परिवर्तनहरू लागू गर्नुहोस्।
3. परिवर्तनहरू रिमोटमा push गरेर कोड समीक्षाका लागि मर्ज अनुरोध खोल्नुहोस्।
4. समीक्षापछि main शाखामा मर्ज गर्नुहोस्।

**फाइदाहरू:**

* प्रत्येक सुविधा वा सुधारलाई छुट्टै ट्र्याक गर्न सकिन्छ।
* स्थिरता सुनिश्चित गर्न विस्तृत समीक्षा प्रक्रिया।
* CI/CD एकीकरणका लागि उपयोगी।

**कमजोरीहरू:**

* जटिल शाखा व्यवस्थापन हुन सक्छ।
* समन्वय गर्न समय लाग्न सक्छ।

**3. फोर्किङ कार्यप्रवाह (Forking Workflow)**

यो कार्यप्रवाह खुला स्रोत परियोजनाहरूमा बढी प्रचलित छ। कुनै पनि प्रयोगकर्ताले मुख्य भण्डारणलाई **फोर्क (Fork)** गरेर आफ्नो संस्करण बनाउँछन्, परिवर्तन गरेपछि मर्ज अनुरोध पठाउँछन्।

**प्रक्रिया:**

1. मुख्य भण्डारणलाई फोर्क गर्नुहोस्।
2. फोर्क भण्डारणमा परिवर्तनहरू लागू गर्नुहोस्।
3. परिवर्तनहरू फोर्क भित्र push गर्नुहोस्।
4. मुख्य भण्डारणमा मर्ज अनुरोध पठाउनुहोस्।

**फाइदाहरू:**

* मूल परियोजना सुरक्षित रहन्छ, किनकि कोहीले प्रत्यक्ष main शाखामा परिवर्तन गर्न सक्दैन।
* बाहिरी योगदानकर्ताहरूका लागि उपयुक्त।
* अधिक स्वतन्त्रता र व्यक्तिगत प्रयोगकर्ताका लागि राम्रो।

**कमजोरीहरू:**

* समीक्षाका लागि धेरै स्रोतहरू आवश्यक पर्दछ।
* परियोजना मर्मत गर्न अधिक समय लाग्न सक्छ।

**4. गठबन्धन कार्यप्रवाह (Integration Manager Workflow)**

यस कार्यप्रवाहमा, विकासकर्ताहरूले आफ्नै भण्डारणमा परिवर्तन गर्छन्, तर कोडलाई मुख्य शाखामा पठाउनु अघि **एक समावेशकर्ता (Integrator)** द्वारा समीक्षा गरिन्छ।

**प्रक्रिया:**

1. विकासकर्ताले परिवर्तनहरू आफ्नै शाखामा गर्छन्।
2. समावेशकर्ताले विभिन्न योगदानहरू संयोजन गर्छन्।
3. मुख्य भण्डारणमा परिवर्तनहरू मर्ज गरिन्छ।

**फाइदाहरू:**

* परियोजना सुदृढ र राम्रो व्यवस्थापन हुन सक्छ।
* गुणस्तरीय समीक्षा प्रक्रिया।

**कमजोरीहरू:**

* समन्वय र मर्ज प्रक्रियामा धेरै समय लाग्न सक्छ।

**उपयुक्त कार्यप्रवाह कसरी चयन गर्ने?**

तपाईंको टोली वा परियोजनाका आवश्यकताहरूका आधारमा, निम्न आधारहरूमा कार्यप्रवाह चयन गर्न सकिन्छ:

* **सानो टोली (Small Team):** केन्द्रित कार्यप्रवाह उपयुक्त।
* **द्रुत विकास (Fast-Paced Development):** फीचर शाखा कार्यप्रवाह।
* **खुला स्रोत (Open Source):** फोर्किङ कार्यप्रवाह।
* **ठूलो टोली (Large Team with Complex Projects):** गठबन्धन कार्यप्रवाह।

**निष्कर्ष**

Git को वितरित स्वभावले विकासकर्ताहरूलाई अधिक स्वतन्त्रता र लचिलोपन प्रदान गर्छ। टोली वा परियोजनाको प्रकृतिअनुसार सही कार्यप्रवाह चयन गर्नु, दीर्घकालीन सफलता सुनिश्चित गर्न अत्यन्त आवश्यक छ।

केन्द्रित कार्यप्रवाह (Centralized Workflow)

केन्द्रित संस्करण नियन्त्रण प्रणालीहरू (CVCS) जस्तै Subversion (SVN) मा, एउटा **केन्द्रिय भण्डारण (Central Repository)** हुन्छ, जसले सम्पूर्ण कोड परिवर्तनहरू स्वीकार्छ। सबै विकासकर्ताहरूले यही भण्डारणसँग समक्रमण (synchronize) गरेर कार्य गर्छन्। Git मा पनि यसै प्रकारको केन्द्रित कार्यप्रवाह सहज रूपमा प्रयोग गर्न सकिन्छ।

**कार्यप्रवाहको प्रक्रिया (Workflow Process)**

1. **क्लोन (Clone):**
   * सबै विकासकर्ताहरूले एकै केन्द्रिय भण्डारणबाट git clone गरेर आफ्नो स्थानीय भण्डारण बनाउँछन्।
2. **परिवर्तन (Changes):**
   * प्रत्येक विकासकर्ताले आफ्नो स्थानीय शाखामा परिवर्तन गर्छन्।
3. **पुश (Push):**
   * पहिलो विकासकर्ताले आफ्नो परिवर्तनहरू git push मार्फत केन्द्रिय भण्डारणमा पठाउँछन्।
   * दोश्रो विकासकर्ताले यदि केन्द्रिय भण्डारणमा परिवर्तन भइसकेको छ भने, उनलाई पहिले ती परिवर्तनहरू git pull मार्फत मर्ज (merge) गर्नु आवश्यक हुन्छ।
4. **फेच र मर्ज (Fetch and Merge):**
   * यदि कुनै विकासकर्ताले आफ्नो परिवर्तनहरू केन्द्रिय भण्डारणमा पठाउन खोज्दा बीचमा अरू विकासकर्ताले कोड पठाइसकेका छन् भने, उनीहरूले git fetch गरेर पछिल्लो परिवर्तन मर्ज गर्नु पर्छ।
   * मर्ज प्रक्रिया सफल भएपछि मात्र उनीहरू आफ्नो परिवर्तन git push मार्फत पठाउन सक्छन्।

**A diagram of a software company

Description automatically generated**

**उदाहरण (Example)**

मध्यस्थ भण्डारणमा काम गर्ने क्रममा, मानौं कि **John** र **Jessica** सँग एउटै रिमोट भण्डारण छ।

1. **John को कार्य:**
   * John ले परिवर्तन गर्यो।
   * git commit गरेर git push गर्यो।
2. **Jessica को कार्य:**
   * Jessica ले पनि परिवर्तन गरिन् तर उनले git push गर्दा त्रुटि (error) आयो।
   * Git ले उनीलाई "non-fast-forward" समस्या देखायो र "pull and merge" गर्न भन्यो।
   * Jessica ले git pull गरेर John को परिवर्तन मर्ज गरिन्।
   * अन्ततः उनले git push गरिन्।

**केन्द्रित कार्यप्रवाहका फाइदाहरू (Advantages)**

* **परिचितता (Familiarity):**
  + अन्य केन्द्रित प्रणालीहरूसँग तुलना गर्दा, यो कार्यप्रवाह धेरै विकासकर्ताहरूका लागि परिचित र सजिलो हुन्छ।
* **सरल व्यवस्थापन (Simple Management):**
  + केवल एकल रिमोट भण्डारणमा सबै परिवर्तनहरू समावेश हुने भएकाले व्यवस्थापन सरल हुन्छ।
* **सहयोग (Collaboration):**
  + साना वा ठूला टोलीहरूका लागि, साझा भण्डारण प्रयोग गरेर सजिलै समन्वय गर्न सकिन्छ।
* **शाखा समर्थन (Branching Support):**
  + Git को शाखा प्रणालीले समान भण्डारणमा दर्जनौं शाखाहरू बनाएर काम गर्न मद्दत गर्छ।

**केन्द्रित कार्यप्रवाहका सीमाहरू (Limitations)**

* **संघर्ष समाधान (Conflict Resolution):**
  + यदि धेरै विकासकर्ताहरू एउटै फाइलमा परिवर्तन गर्छन् भने, मर्ज गर्दा द्वन्द्व (conflict) उत्पन्न हुन सक्छ।
* **सर्वर निर्भरता (Server Dependency):**
  + यदि केन्द्रिय भण्डारण उपलब्ध छैन भने, विकासकर्ताहरू नयाँ परिवर्तनहरू साझा गर्न सक्दैनन्।
* **सीमित मापन योग्यता (Limited Scalability):**
  + ठूलो परियोजनामा समन्वय गर्न कठिन हुन सक्छ, किनकि सबै परिवर्तनहरू एउटै केन्द्रिय स्थानमा निर्भर हुन्छन्।

**निष्कर्ष (Conclusion)**

यदि तपाईंको टोली पहिले नै केन्द्रित संस्करण नियन्त्रण प्रणालीहरूमा काम गरिरहेको छ भने, Git को केन्द्रित कार्यप्रवाह सजिलै अपनाउन सकिन्छ। एकल रिमोट भण्डारणमा परिवर्तन समक्रमण गर्ने सरल विधि भएकाले, यो प्रविधि **सानो वा मध्यम** टोलीहरूका लागि विशेष रूपमा उपयुक्त छ।

**महत्त्वपूर्ण आदेशहरू:**

# रिपोजिटरी क्लोन गर्नुहोस्

git clone <repository\_url>

# परिवर्तनहरू प्रतिबद्ध गर्नुहोस्

git commit -m "Your message"

# रिमोट रिपोजिटरीमा परिवर्तनहरू पठाउनुहोस्

git push origin main

# रिमोटमा भएका परिवर्तनहरू ल्याउनुहोस्

git pull origin main

यस कार्यप्रवाहले विकासकर्ताहरूलाई एकै भण्डारणमा प्रभावकारी रूपमा सहकार्य गर्न सक्षम बनाउँछ, तर ठूलो परियोजनाहरूका लागि **वितरित कार्यप्रवाह (Distributed Workflow)** उपयुक्त हुन सक्छ।

इंटीग्रेशन-मैनेजर वर्कफ़्लो (Integration-Manager Workflow)

Git की **वितरित प्रकृति (distributed nature)** के कारण, यह संभव है कि प्रत्येक डेवलपर के पास अपनी स्वयं की **सार्वजनिक रिपॉज़िटरी (public repository)** में **लिखने (write)** की अनुमति हो और दूसरों की रिपॉज़िटरी में **पढ़ने (read)** की अनुमति हो। इस वर्कफ़्लो में आमतौर पर एक **आधिकारिक (official)** रिपॉज़िटरी होती है जिसे परियोजना का मुख्य स्रोत माना जाता है।

**वर्कफ़्लो की प्रक्रिया (Workflow Process)**

1. **प्रोजेक्ट मेंटेनर का कार्य:**
   * प्रोजेक्ट मेंटेनर (maintainer) अपने सार्वजनिक रिपॉज़िटरी में कोड पुश करता है।
2. **योगदानकर्ता (Contributor) का कार्य:**
   * योगदानकर्ता मेंटेनर की रिपॉज़िटरी को git clone करके अपनी लोकल कॉपी बनाता है।
   * आवश्यक परिवर्तन करने के बाद, वे अपनी खुद की सार्वजनिक रिपॉज़िटरी में कोड पुश करते हैं।
3. **पुल अनुरोध (Pull Request):**
   * योगदानकर्ता मेंटेनर को ईमेल या प्लेटफ़ॉर्म (GitHub/GitLab) पर **pull request** भेजता है।
4. **मेंटेनर प्रतिक्रिया:**
   * मेंटेनर योगदानकर्ता की रिपॉज़िटरी को **रिमोट** के रूप में जोड़ता है और इसे अपनी लोकल शाखा में मर्ज करता है।
5. **मुख्य रिपॉज़िटरी में परिवर्तन:**
   * मेंटेनर परीक्षण (testing) के बाद, संशोधन को आधिकारिक रिपॉज़िटरी में पुश करता है।

**A diagram of a software developer

Description automatically generated**

**उदाहरण (Example)**

मान लें कि एक प्रोजेक्ट में **Alice (मेंटेनर)** और **Bob (योगदानकर्ता)** हैं।

1. **Alice** ने GitHub पर एक सार्वजनिक रिपॉज़िटरी बनाई और Bob ने इसे git clone किया।
2. **Bob** ने परिवर्तन किए और इसे अपने फोर्क में git push कर दिया।
3. Bob ने Alice को pull request भेजा।
4. Alice ने Bob की रिपॉज़िटरी को रिमोट के रूप में जोड़ा:

git remote add bob\_repo https://github.com/bob/project.git

1. Alice ने Bob के बदलावों को मर्ज किया और फिर अपनी आधिकारिक रिपॉज़िटरी में पुश कर दिया।

git pull bob\_repo main

git push origin main

**इंटीग्रेशन-मैनेजर वर्कफ़्लो के लाभ (Advantages)**

* **स्वतंत्र कार्य (Independent Work):**
  + योगदानकर्ता और मेंटेनर अपने-अपने समय के अनुसार काम कर सकते हैं।
  + मेंटेनर तब तक बदलावों को स्वीकार करने के लिए बाध्य नहीं होते जब तक वे तैयार न हों।
* **खुले योगदान (Open Contribution):**
  + कोई भी डेवलपर अपनी सार्वजनिक कॉपी में बदलाव कर सकता है और आवश्यकतानुसार अनुरोध भेज सकता है।
* **बेहतर गुणवत्ता नियंत्रण (Better Quality Control):**
  + मेंटेनर कोड की समीक्षा और परीक्षण करने के बाद ही बदलाव को स्वीकार करता है।
* **कोड सुरक्षा (Code Security):**
  + योगदानकर्ता को मुख्य रिपॉज़िटरी में सीधे पुश करने की अनुमति नहीं होती, जिससे अनावश्यक गड़बड़ी नहीं होती।
* **GitHub/GitLab के साथ आसान एकीकरण (Easy Integration with GitHub/GitLab):**
  + प्लेटफ़ॉर्म में अंतर्निर्मित पुल अनुरोध सिस्टम है, जिससे सहयोग करना आसान हो जाता है।

**सीमाएँ (Limitations)**

* **जटिलता (Complexity):**
  + नए डेवलपर के लिए कार्यप्रवाह समझना थोड़ा कठिन हो सकता है।
* **मर्ज करने में समय (Merge Delays):**
  + यदि मेंटेनर सक्रिय नहीं है, तो योगदानकर्ता को लंबे समय तक प्रतिक्रिया के लिए इंतजार करना पड़ सकता है।
* **रिपॉज़िटरी प्रबंधन (Repository Management):**
  + यदि कई योगदानकर्ता हैं, तो मेंटेनर को कई pull requests को मैनेज करना चुनौतीपूर्ण हो सकता है।

**निष्कर्ष (Conclusion)**

इंटीग्रेशन-मैनेजर वर्कफ़्लो बड़े और ओपन-सोर्स प्रोजेक्ट्स के लिए अत्यधिक उपयुक्त है, जहाँ कई योगदानकर्ता होते हैं। इससे परियोजना में गुणवत्ता नियंत्रण बना रहता है, और सभी लोग स्वतंत्र रूप से योगदान कर सकते हैं। यदि आपकी टीम में कई डेवलपर हैं और आप सहयोग को बेहतर बनाना चाहते हैं, तो यह वर्कफ़्लो आदर्श है।

**महत्त्वपूर्ण आदेश:**

# मेंटेनर की रिपॉज़िटरी क्लोन करें

git clone https://github.com/maintainer/project.git

# लोकल परिवर्तनों को कमिट करें

git commit -m "Fixed a bug"

# अपनी सार्वजनिक रिपॉज़िटरी में पुश करें

git push origin main

# मेंटेनर को pull request भेजें

# (GitHub/GitLab पर UI के माध्यम से)

इस वर्कफ़्लो का उपयोग करके, परियोजना को व्यवस्थित और सुरक्षित रूप से विकसित किया जा सकता है।

तानाशाह और लेफ्टिनेंट्स वर्कफ़्लो (Dictator and Lieutenants Workflow)

यह वर्कफ़्लो बड़े प्रोजेक्ट्स के लिए उपयुक्त है, जहाँ सैकड़ों योगदानकर्ता होते हैं। इसका एक प्रसिद्ध उदाहरण **Linux कर्नेल** प्रोजेक्ट है। इस वर्कफ़्लो में प्रोजेक्ट का संगठन **तानाशाह (dictator)** और **लेफ्टिनेंट्स (lieutenants)** के रूप में होता है।

**कार्यप्रणाली (Workflow Process)**

1. **डेवलपर्स का कार्य:**
   * डेवलपर्स किसी **टॉपिक ब्रांच (topic branch)** में काम करते हैं।
   * वे अपनी शाखा को **रेफरेंस रिपॉज़िटरी** की मास्टर शाखा पर **रीबेस (rebase)** करते हैं।
   * मास्टर ब्रांच वही होती है जिसे तानाशाह अपडेट करता है।
2. **लेफ्टिनेंट्स का कार्य:**
   * लेफ्टिनेंट्स (जो विभिन्न क्षेत्रों के प्रबंधक हैं) डेवलपर्स की टॉपिक ब्रांच को अपनी मास्टर शाखा में मर्ज करते हैं।
3. **तानाशाह का कार्य:**
   * तानाशाह सभी लेफ्टिनेंट्स की मास्टर शाखाओं को अपनी मास्टर शाखा में मर्ज करता है।
4. **पब्लिक अपडेट:**
   * अंततः तानाशाह मास्टर ब्रांच को **रेफरेंस रिपॉज़िटरी** में पुश करता है।
   * सभी डेवलपर नवीनतम अपडेट को फिर से रीबेस कर सकते हैं।

**A diagram of a company

Description automatically generated**

**उदाहरण (Example)**

मान लें कि एक प्रोजेक्ट में निम्नलिखित भूमिकाएँ हैं:

* **तानाशाह (Dictator):** मुख्य परियोजना प्रबंधक (e.g., Linus Torvalds)
* **लेफ्टिनेंट्स (Lieutenants):** अनुभाग प्रबंधक (e.g., नेटवर्किंग, फाइल सिस्टम आदि)
* **डेवलपर्स (Developers):** फीचर्स पर काम करने वाले

**वर्कफ़्लो की स्टेप्स**

1. **डेवलपर का काम:**

git checkout -b feature\_branch

# कोड परिवर्तन करें

git commit -m "Implemented new feature"

git rebase origin/master

git push origin feature\_branch

1. **लेफ्टिनेंट का काम:**

git checkout master

git merge feature\_branch

git push origin master

1. **तानाशाह का काम:**

git checkout master

git merge lieutenant1/master

git merge lieutenant2/master

git push origin master

**तानाशाह और लेफ्टिनेंट्स वर्कफ़्लो के लाभ (Advantages)**

* **आसान प्रबंधन (Better Management):**
  + परियोजना का नेतृत्व मुख्य तानाशाह करता है, लेकिन कार्य को उप-प्रबंधकों (lieutenants) में विभाजित करके काम की जटिलता को कम किया जाता है।
* **बड़ा प्रोजेक्ट हैंडलिंग (Handling Large Projects):**
  + सैकड़ों डेवलपर्स को समन्वयित करने के लिए उपयुक्त।
* **स्थिरता और गुणवत्ता नियंत्रण (Stability and Quality Control):**
  + सभी बदलावों की जाँच लेफ्टिनेंट और फिर तानाशाह द्वारा की जाती है, जिससे उच्च गुणवत्ता सुनिश्चित होती है।
* **अधिक पारदर्शिता (More Transparency):**
  + डेवलपर्स को स्पष्ट रूप से पता होता है कि कोडबेस में क्या चल रहा है और वे आसानी से अपनी ब्रांच को अपडेट कर सकते हैं।

**सीमाएँ (Limitations)**

* **जटिल संरचना (Complex Structure):**
  + छोटे और तेज़-तर्रार टीमों के लिए यह प्रक्रिया बोझिल हो सकती है।
* **प्रक्रिया में देरी (Delays in Process):**
  + यदि लेफ्टिनेंट्स या तानाशाह व्यस्त होते हैं, तो कोड मर्ज होने में देरी हो सकती है।
* **हायरार्किकल निर्भरता (Hierarchical Dependency):**
  + प्रोजेक्ट की गति मुख्य रूप से तानाशाह के निर्णयों पर निर्भर करती है।

**निष्कर्ष (Conclusion)**

तानाशाह और लेफ्टिनेंट्स वर्कफ़्लो बड़े और जटिल प्रोजेक्ट्स के लिए उपयुक्त है, जहाँ उच्च स्तर का प्रबंधन और गुणवत्ता नियंत्रण आवश्यक है। इसका मुख्य उद्देश्य कार्यभार को विभाजित करके परियोजना की स्थिरता और गुणवत्ता बनाए रखना है।

**महत्वपूर्ण Git आदेश:**

# डेवलपर का कार्य

git checkout -b feature\_branch

git rebase origin/master

git push origin feature\_branch

# लेफ्टिनेंट्स का कार्य

git merge feature\_branch

git push origin master

# तानाशाह का कार्य

git merge lieutenant/master

git push origin master

इस वर्कफ़्लो का पालन करके, एक संगठित और व्यवस्थित विकास प्रक्रिया सुनिश्चित की जा सकती है।

**स्रोत कोड शाखाहरू व्यवस्थापन गर्ने ढाँचाहरू**  
मार्टिन फाउलरले "**Patterns for Managing Source Code Branches**" शीर्षकमा एक विस्तृत मार्गदर्शन तयार गरेका छन्, जसले प्रायः प्रयोग गरिने Git वर्कफ्लोहरूलाई समेट्छ र तिनीहरूलाई कहिले र कसरी प्रयोग गर्ने भन्ने व्याख्या गर्छ।

**Git वर्कफ्लोहरूको परिचय**

Git, एक वितरित संस्करण नियन्त्रण प्रणाली (DVCS) भएकाले, विभिन्न प्रकारका वर्कफ्लोहरू प्रदान गर्दछ। यी वर्कफ्लोहरू विकास प्रक्रियामा **समन्वय आवृत्ति**, सहयोग, र कोड स्थिरता सन्तुलन गर्न केन्द्रित छन्।

**प्रमुख Git वर्कफ्लोहरू**

1. **फिचर ब्रान्च वर्कफ्लो (Feature Branch Workflow)**
   * प्रत्येक नयाँ सुविधाका लागि एक छुट्टै शाखा (branch) सिर्जना गरिन्छ।
   * कोड मुख्य शाखामा विलय (merge) गर्नुअघि परीक्षण गरिन्छ।
   * कडा समीक्षा र परीक्षण आवश्यक परियोजनाहरूका लागि उपयुक्त।
   * उदाहरण:

git checkout -b feature/new-feature

git commit -m "नयाँ सुविधा थपियो"

git push origin feature/new-feature

1. **Git फ्लो वर्कफ्लो (Git Flow Workflow)**
   * संरचित शाखा मोडल जसमा निम्न शाखाहरू हुन्छन्:
     + master – उत्पादनका लागि तयार कोड।
     + develop – चलिरहेको विकास कार्य।
     + feature – व्यक्तिगत फिचरहरू।
     + release – परिक्षण र स्थिरीकरणको लागि।
     + hotfix – आकस्मिक समस्या समाधान।
   * नियमित रिलीज भएका परियोजनाहरूका लागि उपयोगी।
   * उदाहरण:

git flow feature start new-feature

git flow feature finish new-feature

1. **GitHub/GitLab वर्कफ्लो**
   * सरल वर्कफ्लो जसमा main शाखा र फिचर ब्रान्चहरू प्रयोग गरिन्छ।
   * पुल अनुरोध (Pull Requests) मार्फत कोड समावेश गरिन्छ।
   * प्रक्रियाः
     + नयाँ शाखा सिर्जना गर्नुहोस्।
     + कोड परिवर्तनहरू गर्नुहोस् र पठाउनुहोस्।
     + PR बनाएर समीक्षा गर्नुहोस्।
     + main शाखामा विलय गर्नुहोस्।
2. **Trunk-Based Development (मुख्य शाखा आधारित विकास)**
   * मुख्य शाखामा सिधै कार्य गरिन्छ वा छोटो अवधिका ब्रान्चहरू प्रयोग गरिन्छ।
   * CI/CD प्रविधिहरूमा उपयोगी।
   * उदाहरण:

git checkout main

git pull origin main

git commit -m "साना परिवर्तनहरू"

git push origin main

1. **Forking Workflow (फोर्किङ वर्कफ्लो)**
   * प्रत्येक योगदानकर्ताले मूल भण्डारण (repository) को आफ्नो प्रतिलिपि (fork) बनाउँछ।
   * परिवर्तनहरू आफ्नै भण्डारणमा धकेलिन्छ र PR को माध्यमबाट प्रस्ताव गरिन्छ।
   * खुला स्रोत (open-source) परियोजनाहरूका लागि उत्तम।
   * चरणहरू:
     + परियोजना फोर्क गर्नुहोस्।
     + आफ्नो कम्प्युटरमा क्लोन गर्नुहोस्।
     + परिवर्तनहरू गर्नुहोस्।
     + PR पठाउनुहोस्।

**उच्च र न्यून एकीकरण आवृत्तिको तुलना**

* **उच्च एकीकरण आवृत्ति (High Integration Frequency):**
  + बारम्बार कोड एकीकरण गरिन्छ।
  + टकराव कम हुन्छ र सहकार्य वृद्धि हुन्छ।
  + आम रूपमा **Agile** र **CI/CD** विकास वातावरणहरूमा उपयोगी।
* **न्यून एकीकरण आवृत्ति (Low Integration Frequency):**
  + दीर्घकालीन शाखाहरू कम आवृत्तिमा विलय गरिन्छ।
  + गम्भीर टकरावहरू हुनसक्छ।
  + परम्परागत विकास मोडेलहरूमा उपयोगी।

**सही वर्कफ्लो चयन गर्ने तरिका**

तपाईंको टोली र परियोजनाको आवश्यकताहरूमा भर पर्छ:

* **टोलीको आकार:** ठूलो टोलीहरूका लागि Git Flow उपयुक्त हुन्छ, जबकि साना टोलीहरूले Trunk-Based Development अपनाउँछन्।
* **परियोजना प्रकार:** खुला स्रोत परियोजनाहरूका लागि Forking Workflow प्रयोग गरिन्छ।
* **परिनियोजन रणनीति:** CI/CD कार्यान्वयन गर्न उच्च एकीकरण आवृत्तिको प्रयोग गरिन्छ।

थप जानकारीका लागि, हेर्नुहोस्:  
[**मार्टिन फाउलरको मार्गदर्शन**](https://martinfowler.com/articles/branching-patterns.html)

यो मार्गदर्शनले प्रत्येक शाखा व्यवस्थापन रणनीतिको व्याख्या गर्दै यसको व्यावहारिक प्रयोगका बारेमा विस्तृत रूपमा बताउँछ।

END ………..