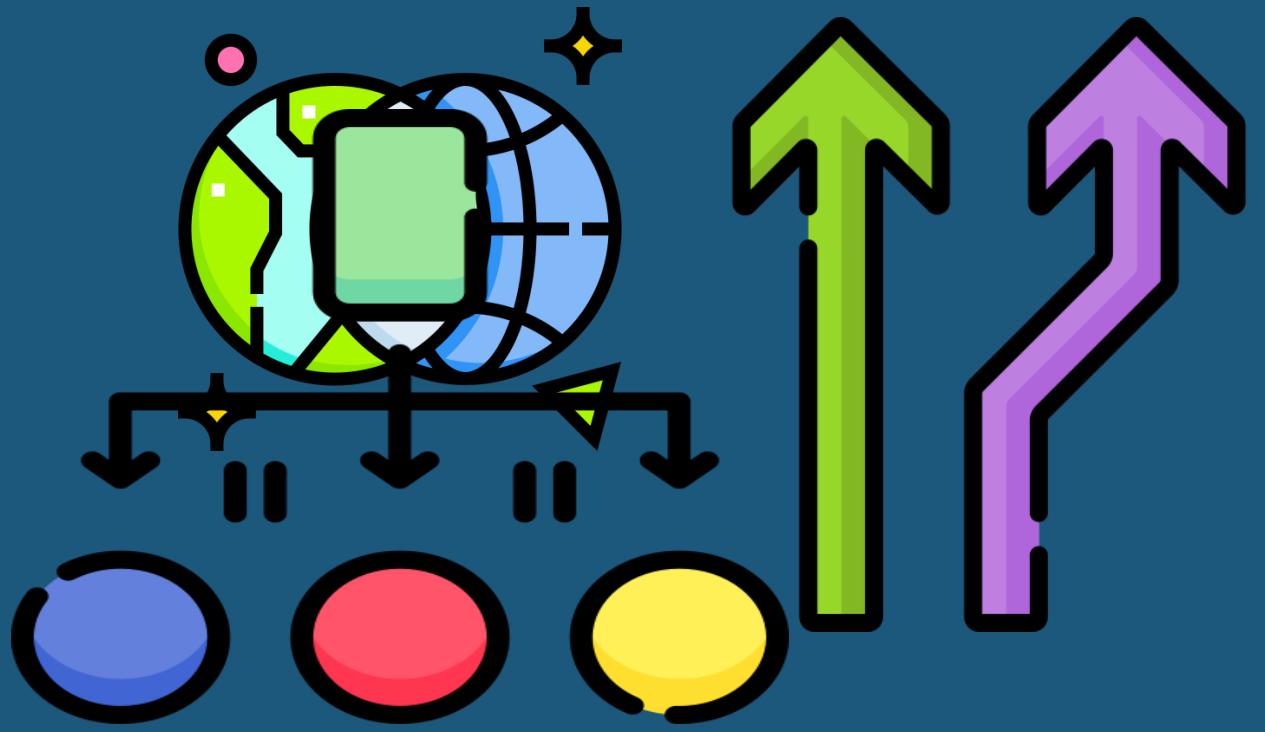


Parallel and Distributed Computing

Mr. Hassan Sardar

Lecturer, Department of Computer Science

COMSATS University Islamabad, Wah Campus

Email: hassan@ciitwah.edu.pk

Lesson

WAH Campus

أَعُوذُ بِاللّٰهِ مِنَ الشَّيْطٰنِ الرَّجِيْمِ

بِسْمِ اللّٰهِ الرَّحْمٰنِ الرَّحِيْمِ

فَكَفٰى بِاللّٰهِ شَهِيْدًا بَيْنَنَا وَ بَيْنَكُمْ اِنْ كُنَّا عَنْ عِبَادَتِكُمْ لَغٰفِلِيْنَ ۝ هُنَالِكَ تَبْلُوْا كُلُّ نَفْسٍ مَّآ اَسْلَفَتْ وَ رُدُّوْٓا اِلَى اللّٰهِ مَوْلٰىهُمُ الْحَقِّ وَ ضَلَّ عَنْهُمْ مَّا كَانُوْا يَفْتَرُوْنَ ۝

سُوْرَةُ يُوْنُس

ہمارے اور تمہارے درمیان اللہ کی گواہی کافی ہے کہ (تم اگر ہماری عبادت کرتے بھی تھے تو) ہم تمہاری اس عبادت سے بالکل بے خبر تھے"۔ اُس وقت ہر شخص اپنے کیے کا مزا چکھ لے گا، سب اپنے حقیقی مالک کی طرف پھیر دیے جائیں گے اور وہ سارے جھوٹ جو انہوں نے گھڑ رکھے تھے گم ہو جائیں گے۔

Today' Lecture

WAH Campus

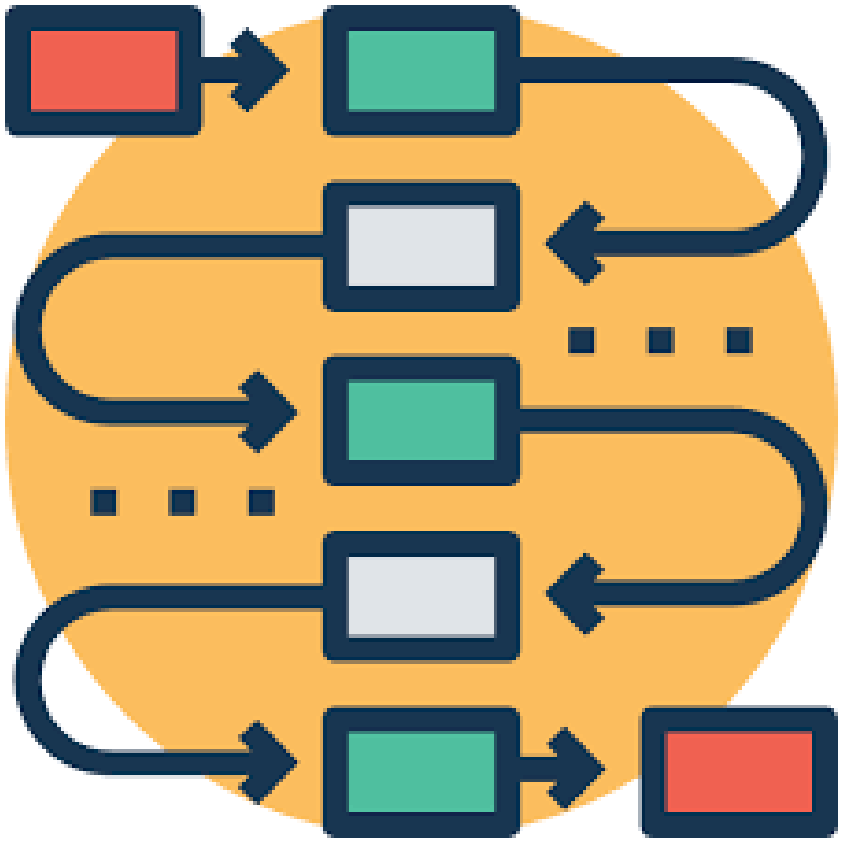
Planning for parallelization

- Planning steps for a parallel project
- Version control and team development workflows
- Understanding performance capabilities and limitations
- Developing a plan to parallelize a routine

Planning steps for a parallel project

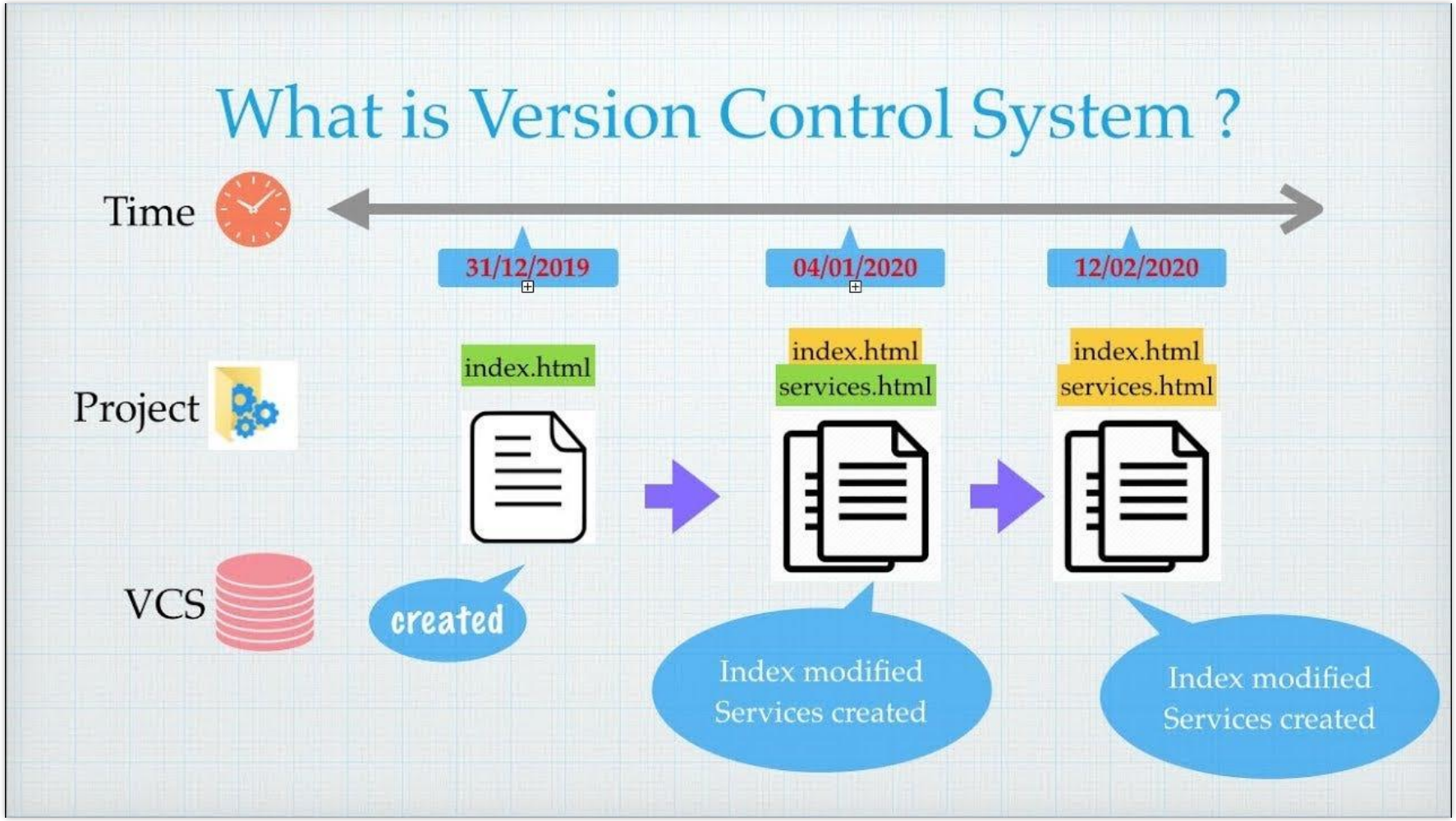
WAH Campus

- Understand the Problem and Application Requirements:
- Analyze the Existing Serial Code:
- Set up Version Control:
- Develop a Test Suite:
- Establish Code Quality and Portability Standards:
- Assess System and Performance Requirements:
- Design a Parallelization Strategy:
- Create a Detailed Project Plan:
- Prepare for Iterative Development:



Version Control System

WAH Campus

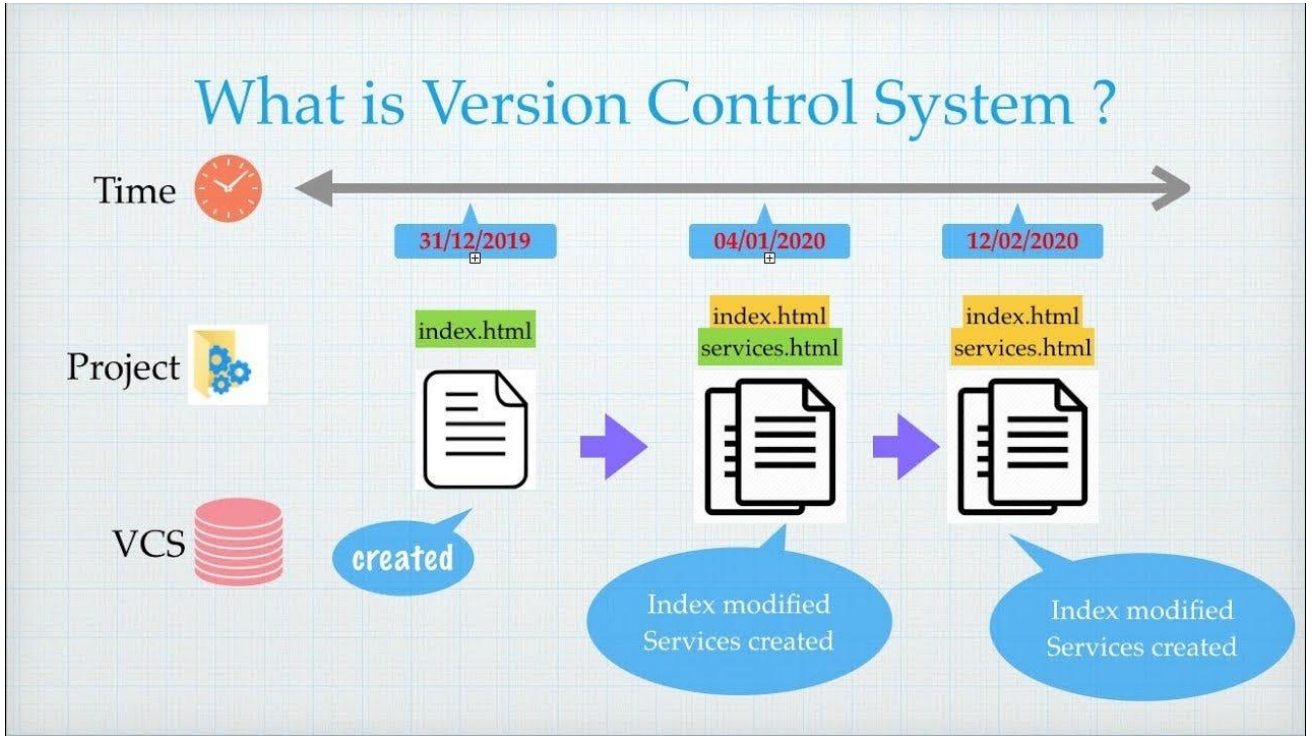


Version control and team development workflows

WAH Campus

Importance of Version Control in Parallel Projects

- **Track Changes**: Version control systems (VCS) allow developers to track every change made to the codebase, making it easy to revert to previous versions if necessary.
- **Collaboration**: Multiple team members can work simultaneously on different parts of the project without overwriting each other's work.
- **Backup and Recovery**: Provides a backup of the codebase, allowing for easy recovery in case of errors or corruption.



Version control and team development workflows

WAH Campus

Popular Version Control Systems (VCS)

Git: Most widely used distributed VCS. Supports offline commits and flexible workflows.

SVN (Subversion): A centralized VCS, suitable for projects where tight control and security are paramount.

Mercurial: Similar to Git but often preferred for projects with larger codebases.

Perforce / ClearCase: Commercial tools often used for enterprise-scale projects.

Version control and team development workflows

WAH Campus

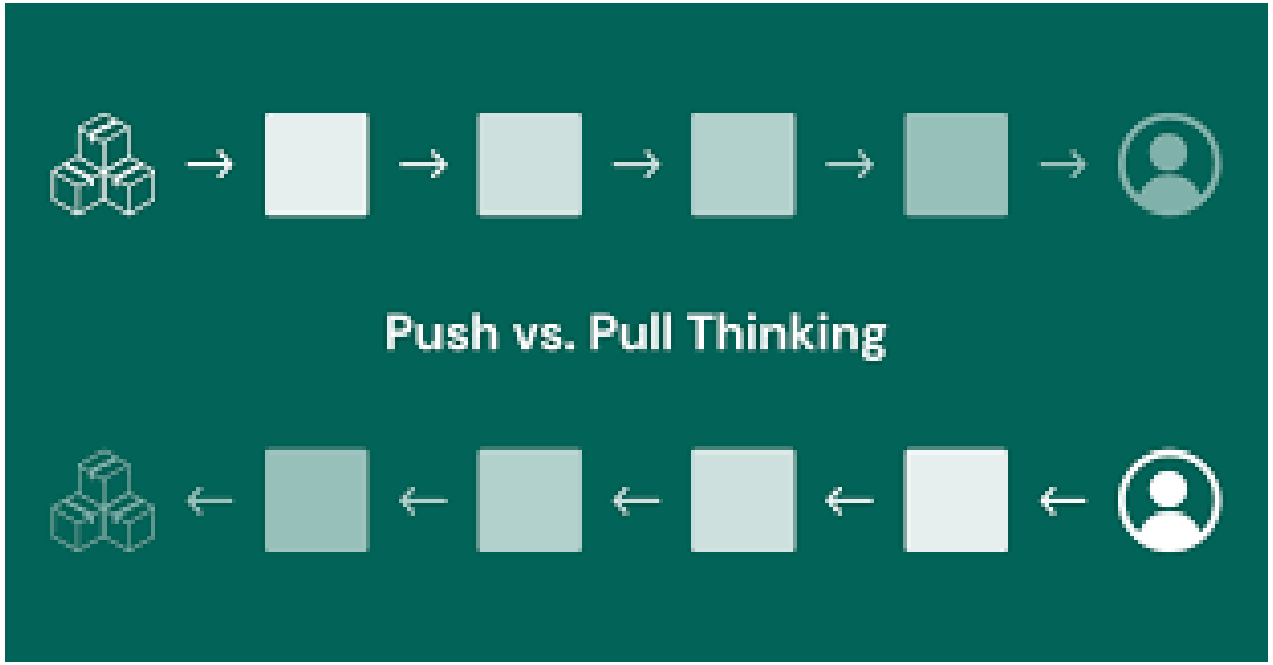
Development Workflows in Version Control

Push Model (Direct Commit):

- Developers commit directly to the main branch.
- Fast and easy for small teams or incremental parallel tasks.
- **Challenges**: Less oversight, prone to errors if there are frequent changes.

Pull Request (Merge Request) Model:

- Developers create branches for new features or fixes and submit them for review.
- Other team members review and approve before merging into the main branch.



Key Practices for Effective Version Control in Parallel Projects

WAH Campus

Frequent Commits: Commit small, incremental changes often to avoid large, unmanageable diffs and to aid debugging.

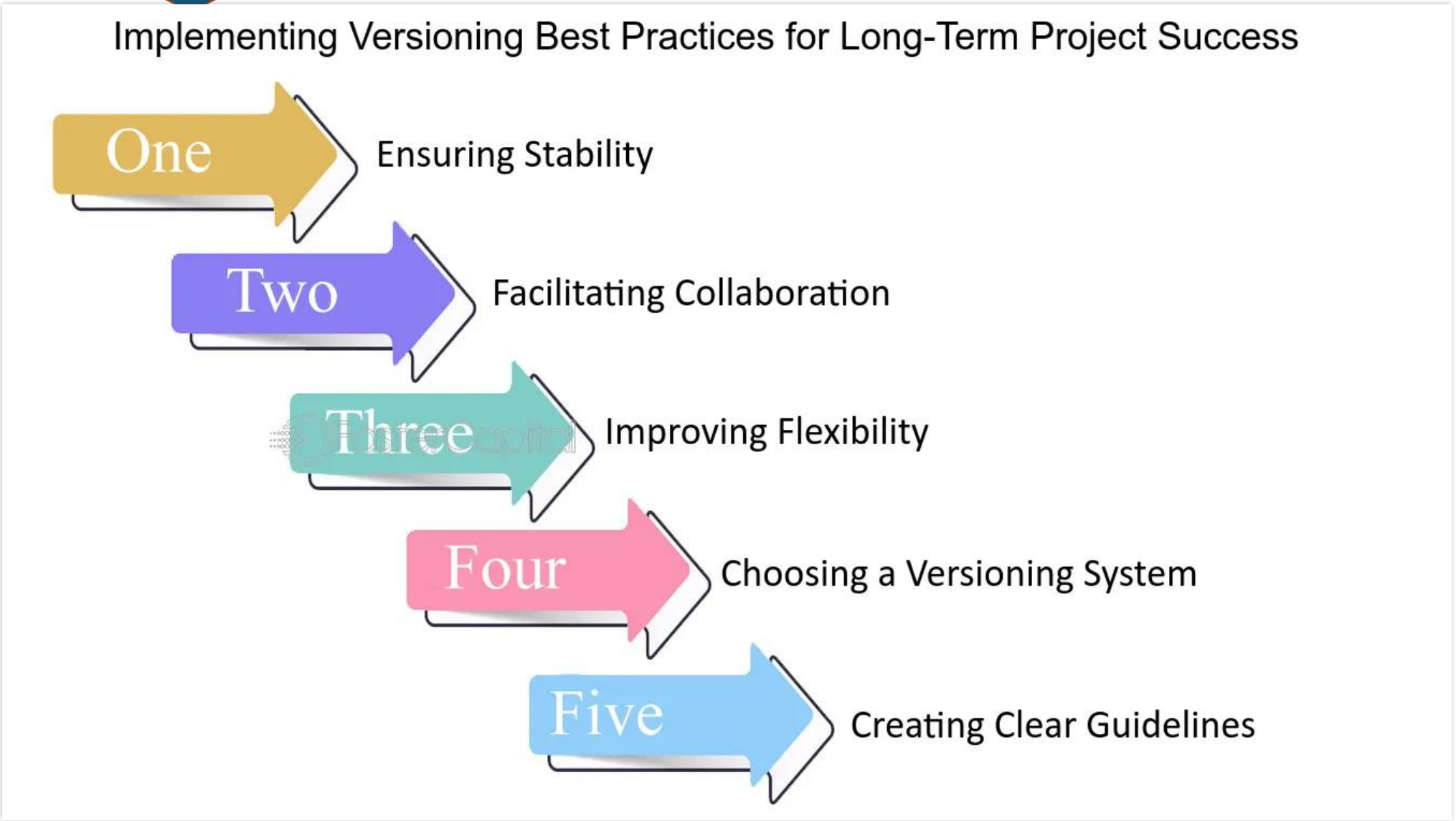
Branching Strategy: Use feature branches, bug fix branches, and release branches to organize the workflow.

Commit Messages: Write detailed commit messages that clearly explain the changes and their purpose (e.g., "Fixed race condition in OpenMP loop").

Tagging and Versioning: Use tags for marking stable releases or significant milestones in the development process.

Key Practices for Effective Version Control in Parallel Projects

WAH Campus



Example Workflow

Create a Branch: Developer creates a new branch for a parallelization task.

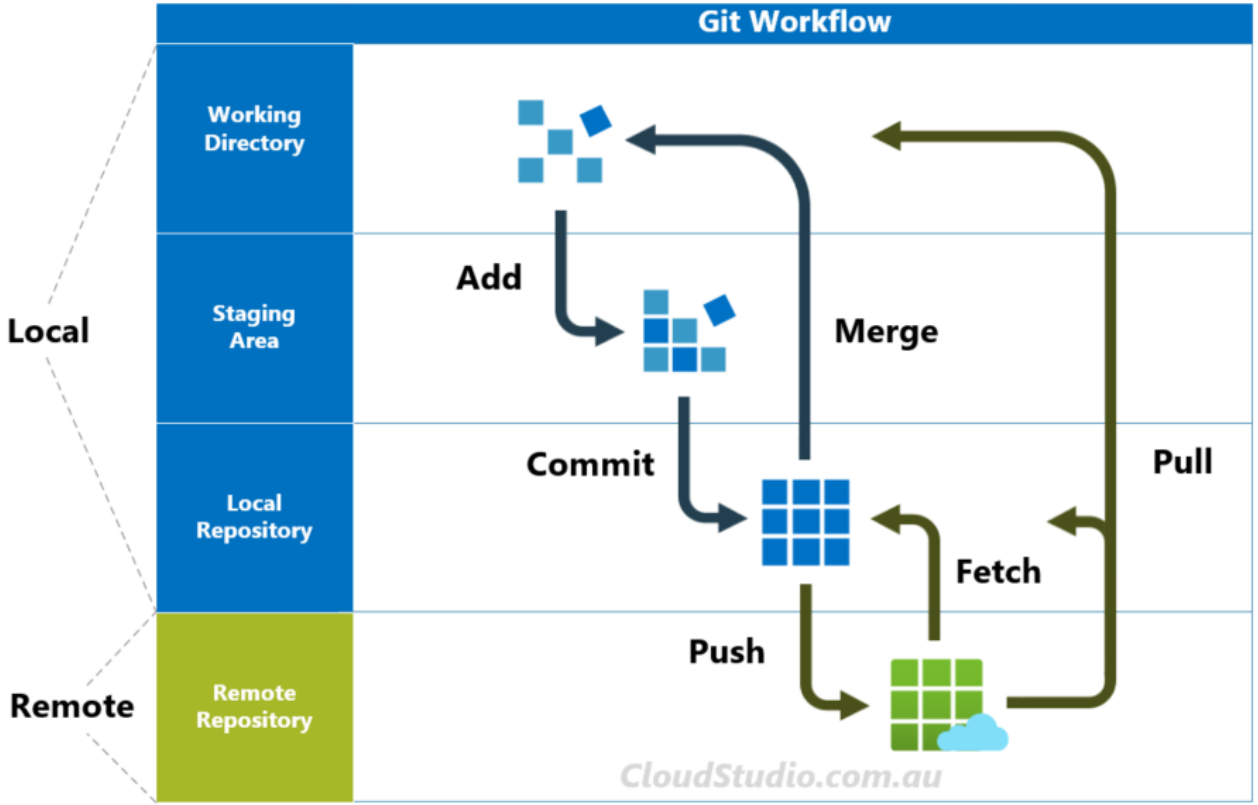
Develop in Branch: Code is written, tested, and committed regularly in the branch.

Submit Pull Request: Once development is complete, a pull request is submitted for review.

Code Review: Team members review the code for quality, functionality, and performance.

Merge to Main Branch: Once approved, the code is merged, triggering tests in the CI pipeline.

Continuous Integration: Automated tests verify that the changes do not break existing functionality.



Continuous Integration with Version Control

WAH Campus

Automated Testing: Integrate testing frameworks with the version control system to automatically run tests after each commit or pull request.

Continuous Integration (CI) Tools: Tools like Jenkins, Travis CI, GitLab CI, and CircleCI automatically build and test code after changes are made.

Benefits: CI helps catch errors early, ensuring code remains stable and functional during parallel development.

Understanding performance capabilities and limitations

WAH Campus

- **Increased Speed (throughput)**:Parallel processing can significantly reduce execution time by dividing tasks into smaller, independent sub-tasks, allowing them to be executed simultaneously. This is particularly effective in compute-bound applications like simulations, video encoding, and scientific calculations.
- **Better Utilization of Multi-core Processors**: Modern processors have multiple cores, and parallelization can leverage these to maximize the system's computational power. For instance, a quad-core processor could theoretically run four tasks simultaneously.
- **Scalability**: Parallelization allows an application to scale with the number of available processing units (e.g., cores in a CPU, nodes in a cluster). In distributed systems, parallelization can span across multiple machines.
- **Enhanced Responsiveness**: In user-interface (UI) applications, parallelization ensures that intensive background tasks (like file I/O or data processing) don't block the UI thread, enhancing user experience by keeping the application responsive.
- **Optimized Resource Use**:It allows better utilization of available resources (CPU, GPU, memory) by breaking tasks into smaller ones that can be processed concurrently, avoiding idle times.

Understanding performance capabilities and limitations

- **Amdahl's Law**: This principle highlights the diminishing returns of parallelization. It states that the speedup gained by parallelizing a task is limited by the fraction of the task that cannot be parallelized. Even with infinite processors, the non-parallelizable portion limits maximum speedup.
- **Overhead**: Parallelization introduces communication, synchronization, and context-switching overhead. Dividing a task, managing threads, and sharing data between them incurs performance costs. If these overheads are too large, parallelization can even degrade performance
- **Data Dependency**: If tasks are interdependent or require shared data, synchronization becomes necessary, which can cause bottlenecks (e.g., locking mechanisms, race conditions). These dependencies reduce the efficiency of parallel execution.
- **Load Balancing**: Uneven distribution of tasks can lead to some processors being overworked while others remain idle. Effective load balancing is critical to maximize the benefits of parallelization, especially in heterogeneous or distributed systems.
- **Scalability Limits**: While parallelization can improve scalability, it has limits. Eventually, adding more processing units may not improve performance due to factors like memory access contention, synchronization delays, and diminishing task division returns.

Understanding performance capabilities and limitations

Example:

- **Embarrassingly Parallel Tasks**: Tasks like rendering individual frames in a video can be parallelized almost perfectly since they don't depend on each other.
- **Non-parallelizable Tasks**: Operations like updating a global variable shared by all threads will involve locks and synchronization, significantly reducing parallelization effectiveness.

- Understanding these performance capabilities and limitations is critical for designing efficient parallel systems and making informed decisions about when and how to parallelize tasks.

Developing a plan to parallelize a routine

- **Identify the Routine:** First, define the routine you want to parallelize. It could be a loop, a function, or an entire algorithm. This routine should have elements that can be divided into smaller tasks that may run independently.
- **Analyze for Parallelism:** Examine the routine to determine whether it can be parallelized. Key factors to consider include:
  - **Independent tasks**: Can parts of the routine be executed independently, or are there dependencies between tasks?
  - **Task granularity**: Will the tasks created by parallelization be substantial enough to justify the overhead?
  - **Potential bottlenecks**: Are there shared resources (e.g., variables or memory) that may cause race conditions or require synchronization?
- **Choose a Parallelization Strategy**
  - Select an appropriate parallelization model depending on the nature of the task:
  - **Task Parallelism**: Where different tasks or functions can be run simultaneously. Good for heterogeneous processes (e.g., each core working on a different function).
  - **Data Parallelism**: Where the same operation is performed on different pieces of data. Suitable for repetitive operations on large datasets, like matrix multiplications or image processing.
  - **Pipeline Parallelism**: Where different stages of a process can be performed concurrently, ideal for stream processing.
  - **Fork-Join Parallelism**: Where a routine forks into multiple parallel tasks and joins again at the end, common in recursive algorithms.

Developing a plan to parallelize a routine

WAH Campus

- **Decompose the Routine**

Break the routine into smaller, independent sub-tasks. This could involve:

- ✓ **Loop splitting**: Dividing loop iterations into chunks that can run independently.
- ✓ **Function splitting**: Isolating independent parts of a function.
- ✓ **Recursive splitting**: In divide-and-conquer algorithms (e.g., quicksort, mergesort), dividing the problem into subproblems and solving them in parallel.

- **Choose the Right Parallelization Tools/Frameworks:**
- **Implement Parallelization:**
- **Optimize and Minimize Overhead:** Avoid unnecessary synchronization, Minimize inter-process communication:
- **Test for Correctness:** All tasks complete correctly, Shared data is correctly synchronized.
- **Measure Performance , Refine and Optimize**