

alok.giri@ncit.edu.np

Traditional Methodologies

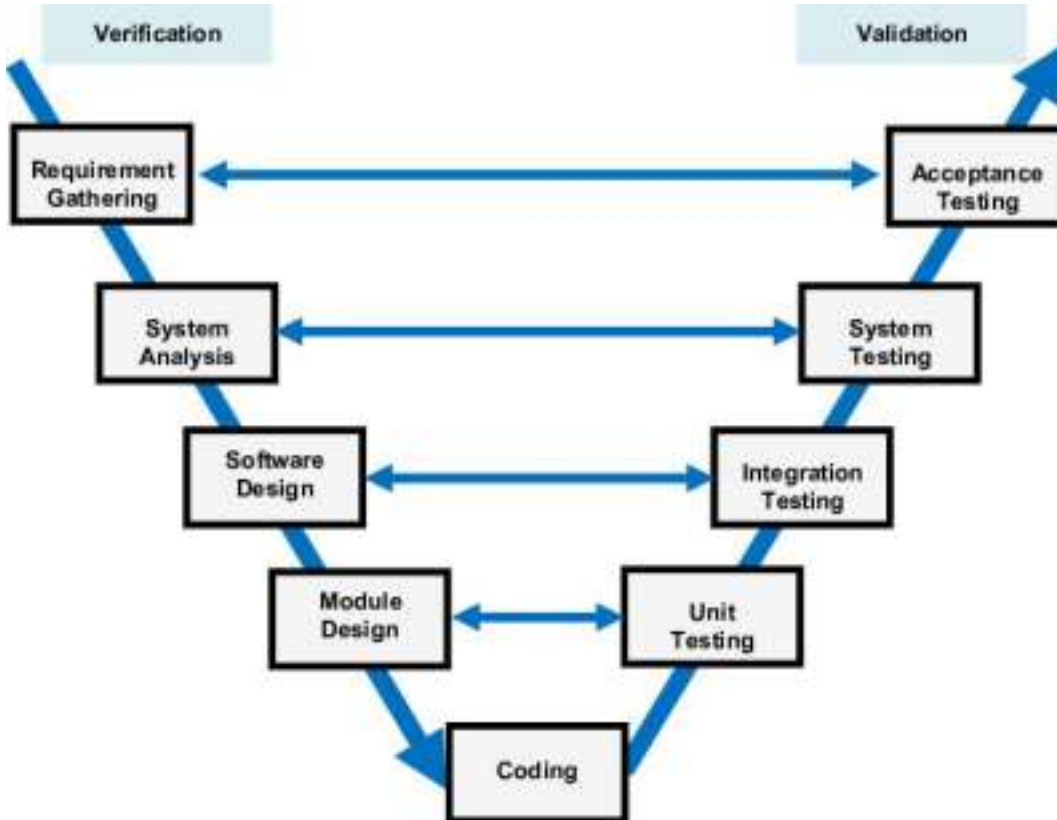


Waterfall: Linear, rigid, one phase must finish before the next.

SPIRAL MODEL IN SOFTWARE DEVELOPMENT



Spiral: Combines Waterfall with iterative refinement;



adds risk analysis at every loop

V-Model: Emphasizes Verification and Validation at each state

Limitations of Traditional

Methods

- These models assume that requirements don't change. But in reality, customers often don't know exactly what they want until they see a working project.
- Late testing = late feedback = higher cost - Too much documentation, too little working product

**Why not break work into
smaller parts and deliver every**

2-4 weeks? That's the idea behind iterative development.

- Introduction to Agile Mindset

- Agile emerged in response to frustration with slow, inflexible models.
- Agile values collaboration, continuous delivery, adaptability.

Agile is not just a process– it's a mindset shift. It says, 'Expect change, welcome feedback and deliver value fast'

- **Need for Flexibility and Rapid Delivery** -

Market trends, startups and tech evolution required faster releases.

- Agile supports customer collaboration and shorter release cycles.

Amazon deploys code every 11.6 seconds. That's not possible with Waterfall.

Agile Manifesto

Four Agile Values

- Individuals & Interactions over Process and Tools
- Working Software over comprehensive documentation
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

Agile Principles

- Early and continuous delivery of valuable software -
Welcome changing requirements, even late in development

- Deliver working software frequently (e.g., every 2 weeks)
- Business and developers must work together daily
- Build projects around motivated individuals - Use face-to-face communication

Agile Principles

- Working software is the primary measure of progress
- Sustainable development pace
- Continuous attention to technical excellence
- Simplicity—the art of maximizing the amount of work not done
- Self-organizing teams produce the best designs

- Teams regularly reflect and adjust

In Agile, retrospectives help teams look back every sprint and improve their process. It's like sports teams watching their match to get better.

Agile vs Traditional Mindset

| Aspect | Traditional Mindset | Agile Mindset |
|----------------|-----------------------------|--------------------------|
| Planning | Fixed upfront | Evolving and adapting |
| Communication | Formal, documentation-heavy | Open and continuous |
| Delivery | End of cycle | Continuous |
| Testing | After coding ends | Parallel and early |
| Success Metric | Meeting plan | Delivering value quickly |

Benefits and Challenges

- Benefits of Agile

- Customer Satisfaction: Frequent delivery of valuable features

- Faster delivery: Small, shippable increments -

- Adaptability: Can change course based on feedback -

- Team Ownership: Empowered and self organizing teams

In Agile, teams feel a sense of ownership. They don't just follow instructions—they solve problems together.

- Challenges in Agile

- Scope Creep: Continuous change without proper control

- Cultural Resistance: Traditional organizations resist flat structures
- Enterprise Policies: Legal and compliance requirements may not match Agile flexibility
- Requires Maturity: Agile fails if people don't collaborate well.

Agile gives freedom– but freedom without discipline leads to chaos.

When to Use Agile

Use Agile when:

- Requirements are likely to change
- Quick feedback is needed

- Innovation is a priority

Avoid Agile when:

- Fixed scope, timeline, and budget
 - Regulatory or safety-critical projects (e.g., aviation software)

alok.giri@ncit.edu.np

What is Scrum?

Scrum is lightweight, Agile framework for managing and completing complex

projects. It is empirical.

Scrum Framework Roles

- **Scrum master**

- Facilitates all Scrum events
 - Ensures team follows scrum principles
 - Protect team from outside distractions -
- Does not act like a boss or traditional manager

Analogy: Like a coach in a football team– not

playing, but helping everyone play better.

- **Product Owner**

- Owns the product vision
- Maintains and prioritizes the Product Backlog - Talks to stakeholders and understands customer needs
- Accepts or rejects work as “Done”

Analogy: Like a restaurant manager who ensures customers are happy and the chefs cook the right dish.

- **Development Team**

- 5–9 people ideally
 - Have all required skills (design, code, test, deploy)
 - Self-organizing: they decide how to do the work -
- No titles or sub-roles inside the team (all are Developers)

Analogy: Like a rock band—each member has a role, but they all own the final performance.

Artifacts

- Product Backlog
 - Owned by Product Owner

- Dynamic and constantly evolving
- Refinement: breaking down, estimating and clarifying items

Example:

.Items: "Login page", "Search filter", "Add to Cart" .

Higher-priority items are better defined

- Sprint Backlog
 - Created during Sprint Planning
 - Developers commit to these items for the sprint
 - Includes tasks and effort estimations

It answers: What can we deliver in the next 2 weeks, and how will we do it?

- Increment

- Must meet Definition of Done (DoD): passed tests, reviewed, deployable
- Delivered to stakeholders during Sprint Review
- Multiple increments may exist, but each increment is additive and complete

Scrum Events

- Daily Stand-up
 - Everyone answers:
 - What did I do yesterday?
 - What will I do today?
 - Any blockers?
 - Encourages team accountability and coordination
 - Same time, same place
- Backlog Refinement

- PO+Dev team meet to:
 - Clarify user stories
 - Estimate effort (story points)
 - Break large items into smaller ones

Backlog refinement avoids surprises in Sprint Planning

- Sprint Planning
 - Sprint goal is defined
 - PO explains top backlog items
 - Dev team selects items they can commit to
 - Tasks are broken down

- Sprint Review
 - Dev team showcases what's "Done"
 - Stakeholders give feedback
 - PO may adjust backlog accordingly

It's not a status meeting. It's about getting real feedback on working software.

- Sprint Retrospective
 - Held after Sprint Review, before next Sprint Planning
 - Only team members + Scrum master

- 3 key questions:
 - What went well?
 - What didn't go well?
 - How can we improve?

Kanban

- Introduction to Kanban
 - Visual system for managing work
 - Helps teams visualize tasks, limit WIP, and improve efficiency
 - Popular in Agile and Lean environments

Kanban was originally developed at Toyota for lean manufacturing. Its goal is simple: visualize how work moves, and continuously improve it.”

- Kanban Principles

- Visualize Workflow

- Limit Work In Progress (WIP)

- Focus on Flow

- Continuous Improvement (Kaizen)

- Visualize Workflow

- Use boards to show task stages (e.g., To Do,

Doing, Done)

- Improves visibility, accountability, and clarity
- Limit Work In Progress (WIP)
 - Set limits on how many tasks are in-progress - Encourages task completion over multitasking - Prevents overload and bottlenecks
- Focus on Flow
 - Monitor how tasks move across the board

- Improve speed and predictability
- Address delays early
- Continuous Improvement
 - Reflect regularly (retrospectives, daily checks)
 - Tweak workflow based on insights
 - Small improvements = big gains over time

Visualizing Workflow

- Create a Kanban Board

- Columns represent stages (e.g., To Do → Design → Dev → QA → Done)
- Swimlanes divide task types (e.g., Bugs, Features)

A Kanban board represents your team's process.

Customize it to your real-life workflow. Swimlanes help manage multiple types of work.

- Kanban Board Components
 - Columns = Workflow stages
 - Cards = Tasks or user stories
 - Swimlanes = Parallel task types
 - Drag cards to indicate progress

Boards are dynamic. As work progresses, cards move right. Teams can instantly see priorities, progress, and blocks.

Managing Work in Progress

- Prevent overload
- Improve focus
- Shorter delivery cycles

If everything is in-progress, nothing is done. By limiting WIP, you free up capacity and increase quality output.

- Monitoring Flow Efficiency
 - Cycle Time: Start to Finish of a task
 - Lead Time: Request to Delivery
 - Shorter times = Better flow

Use these metrics to analyze efficiency. If your cycle time is long, work might be stuck. If your lead time is too long, prioritization might be off.

Extreme Programming (XP)

Extreme Programming (XP) is an Agile software development framework that emphasizes technical

excellence, rapid feedback, and customer involvement.

- XP encourages small, frequent releases - Built for highly dynamic and uncertain projects - Focuses on engineering practices (vs. Scrum's process framework)

XP Values

- Communication – Constant feedback between team members
- Simplicity – Build only what is needed now - Feedback – Get early and frequent feedback (tests,

customers)

- Courage – Refactor, delete code, challenge assumptions
- Respect – Value each team member's input and pace

XP Roles

- Developer- Writes and refactors code, tests -
- Customer- Provides user stories, business goals -
- Tester- Ensures coverage via automated/manual tests -
- Tracker- Monitors progress and velocity
- Coach- Guides the team in XP practices

Practice 1- TDD

- Red: Write a failing test
- Green: Write the minimal code to pass
- Refactor: Clean up code without changing behavior

Benefits

- Instant feedback
- Cleaner code
- Better test coverage
- Encourages simplicity

Practice 2- Pair Programming

TWO BRAINS, ONE KEYBOARD

- 2 developers work together:
 - Driver: Writes the code
 - Navigator: Reviews, thinks ahead, finds bugs
- Switch roles every 30 mins - 1 hour

Benefits: Fewer bugs, shared knowledge, faster problem solving, onboarding becomes easier

Continuous Integration (CI)

- Developers integrate code frequently (multiple times

per day)

- Each integration is verified by automated tests

Benefits:

- Detect integration issues early
- Shortens feedback cycle
- Maintains a working system

Lean Software Development

- What is Lean
 - Origin: Lean manufacturing at Toyota
 - Focus: Maximize value, minimize waste

- In software: Focus on flow, quality, and fast delivery

Lean is about doing more with less

Lean Principles

- Eliminate Waste
 - Waste = anything that doesn't add value to the customer
 - Examples:
 - Extra features no one uses
 - Waiting for approvals
 - Defects and rework

- Too many meetings
- Build Quality In
 - Prevent defects instead of fixing them later
 - Use:
 - Automated testing
 - CI/CD pipelines
 - Code reviews and pair programming
- Defer Commitment
 - Don't make decisions too early
 - Keep options open until the last responsible moment
 - Reduces risk of poor decisions

Booking a flight just in time with enough data vs. too early and getting wrong date

- Deliver Fast
 - Speed = feedback + learning
 - Deliver small, usable chunks
 - Helps validate ideas early

Example: MVP -> deploy fast -> get real feedback

- Respect People
 - Empower teams to make decisions
 - Promote ownership and responsibility

- Avoid micromanagement

Pathao working mechanism.

- Optimize the Whole
 - Don't optimize parts at the cost of the whole system
 - Focus on end-to-end flow, not just development

Dev builds fast but QA gets overwhelmed = not real optimization

- Continuous Improvement (Kaizen)
 - Encourage team to inspect and adapt regularly

- Retrospectives are key
- Encourage experimentation

Eliminating Waste

- What is waste in software?
 - Anything that doesn't directly contribute to delivering value is waste.
- Types of waste:
 - Partially done work, extra features, relearning, handoffs, task switching, delays, defects

Continuous Improvement

- What is Kaizen?
 - Japanese word for “Change for better” -
 - Ongoing, incremental improvements -
 - Driven by everyone, not just management

Practices for Continuous Improvement

- A and B Testing
- User Testing (Beta Testing)
- Team Reviews

alok.giri@ncit.edu.np

Introduction to Agile Planning

- Agile planning is iterative and incremental, focusing on delivering value early and often.
- It involves frequent re-prioritization and adaptive

planning as the project evolves.

- Agile planning typically includes:
 - Release Planning: High-level planning over several iterations.
 - Sprint Planning: Detailed planning for the upcoming sprint.

User Stories

- **Definition:** A user story is a brief, simple description of a feature told from the perspective of the end user.
- **Purpose:** Captures what a user needs and why, helping teams understand user value.

- Example: *As a user, I want to view my transaction history so that I can track my expenses.*

Standard user story template:

As a [type of user], I want [some goal] so that [some reason].

This format helps ensure clarity and alignment with user needs.

INVEST Criteria

Independent: Can be developed and delivered independently.

Negotiable: Not a contract; open to discussion. Valuable:

Delivers value to the customer.

Estimable: Can be estimated for effort.

Small: Can be completed within a sprint.

Testable: Has clear acceptance criteria to validate completion.

Acceptance Criteria

- **Definition:** Specific conditions that must be met for a user story to be considered complete.
- **Format:** Often written using **Given-When-Then** syntax.
 - Given [context],
 - When [action],
 - Then [expected outcome].
- **Example:**

- Given the user is logged in,
- When they click on 'Logout',
 - Then they should be redirected to the login page.

Real-Life Examples of User Stories

1. As a blogger, I want to schedule posts so that I can manage publishing even when I am offline. 2. As a student, I want to receive email reminders for assignment deadlines so that I don't miss them. 3. As an admin, I want to deactivate inactive accounts so that I can maintain security and performance.

Story Points

- **Definition:** A unit of measure to estimate the overall effort needed

to implement a user story.

- Story points are **relative**, not tied to specific hours or days.
- Purpose: Helps measure team velocity and plan sprints accordingly.

Relative Sizing

- Instead of estimating in absolute time, teams compare user stories against each other.
- Helps maintain consistency and reduces estimation errors.
- Techniques include analogy-based comparison and using previously completed stories as reference.

Estimation Factors

When estimating story points, consider:

- **Complexity:** Technical difficulty involved.
 - **Effort:** Amount of work required.
 - **Risk:** Uncertainty or potential issues. ●
- Uncertainty:** Ambiguity in scope or requirements.

Fibonacci Sequence in Estimation

- Common story point values: **1, 2, 3, 5, 8, 13, 21, etc.**
- Larger numbers reflect greater uncertainty. ●

Benefits: Encourages teams to differentiate between simpler and more complex tasks.

Planning Poker

- A collaborative estimation technique where:
 1. A story is presented.
 2. Team members independently select a story point value.
 3. All estimates are revealed simultaneously.
 4. Discussions follow, especially if there are wide discrepancies.
 5. Process repeats until consensus is achieved.
- Promotes team alignment and inclusive decision-making.

Release Planning

- **Definition:** Long-term planning outlining when and what features will be released.
- Involves high-level prioritization of features and estimating how many sprints are needed.
- Aligns business goals with development capacity.

Roadmap Planning

- Strategic plan outlining **product development direction over time**.
- Helps stakeholders visualize the progression of product features.
- Typically divided into **phases** or **quarters**.

Sprint Planning

- Conducted at the beginning of each sprint. ●
- Teams decide **which stories they will commit to delivering** based on priority and capacity.
- Sprint Backlog is created, and each task is discussed.

Release Planning vs Sprint Planning

| Feature Release Planning | Sprint Planning | Focus |
|---------------------------------|------------------------|--------------|
|---------------------------------|------------------------|--------------|

| | | |
|-----------------|------------------|-----------------|
| Long-term goals | Short-term goals | Timeframe |
| Months | 1-4 weeks | Weeks or months |

| |
|-----------------------------|
| Participants |
| Product Owner, Stakeholders |

| |
|------------------|
| Development Team |
|------------------|

Minimum Viable Product (MVP)

- **Definition:** A version of a product with just enough features to satisfy early users and gather feedback.
- **Purpose:** Quickly validate assumptions and minimize development costs.
- **Example:** Dropbox's MVP was a video demo explaining the idea before building the actual product.

Iterative Development

Definition:

- Iterative development is an approach where the project is broken down into small parts and built through repeated cycles (iterations).
- Each iteration involves planning, designing, coding, and testing.

Benefits:

- Continuous feedback from stakeholders.
- Early detection and correction of defects.
- Adaptability to changes.
- Progressive development towards the final product.

Examples:

- Building an e-learning platform: Start with login + dashboard,

then add quizzes, forums, and analytics in future iterations. ●
Agile frameworks like Scrum follow iterative development with **sprints** as iterations.

2. Incremental Delivery

Definition:

- Delivering the product in **increments** or working pieces that add functionality over time.
- Each increment is a **usable** and **potentially shippable** product.

Incremental vs Iterative

| | | |
|----------|-----------|-------------|
| Feature | Iterative | Incremental |
| Approach | Repeating | refinement |

Focus Improvement and Delivery of
feedback usable
functionalities

Example Revise UI
design multiple
times

Deliver
dashboard -> Profile ->
settings

Adding new functional
parts

Benefits:

- Early value delivery to customers.
- Helps prioritize features based on user needs.
- Reduces risk and simplifies testing.

Tracking Progress

Importance:

- Ensures transparency.
- Allows teams to spot issues early.
 - Helps in adapting plans based on actual progress.

Key Tools:

- **Task Boards:** Shows tasks in To Do, In Progress, Done.
- **Kanban Boards:** Visual workflow for continuous delivery.
- **Dashboards:** Central view of metrics, status,

blockers, etc.

Visual Tools

Task Boards:

- Columns: Backlog → To Do → In Progress → Review → Done.
- Great for daily stand-ups and sprint planning.

Kanban Boards:

- Focus on **Work In Progress (WIP) limits**.
- Used for continuous flow rather than time-boxed sprints.
- Excellent for maintenance/support teams.

Dashboards:

- Integrated view of metrics:
 - Velocity
 - Defect count
 - Sprint goal progress
 - Team capacity

Burndown and Burnup Charts

Burndown Chart:

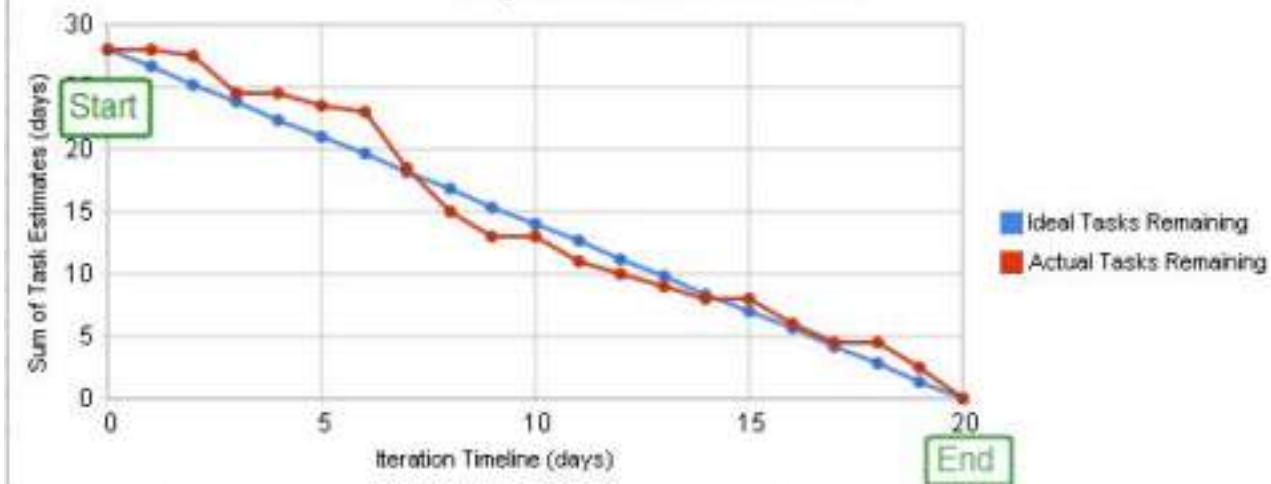
- Tracks remaining work over time.
- X-axis: Time (e.g., sprint days), Y-axis: Remaining effort/story points.

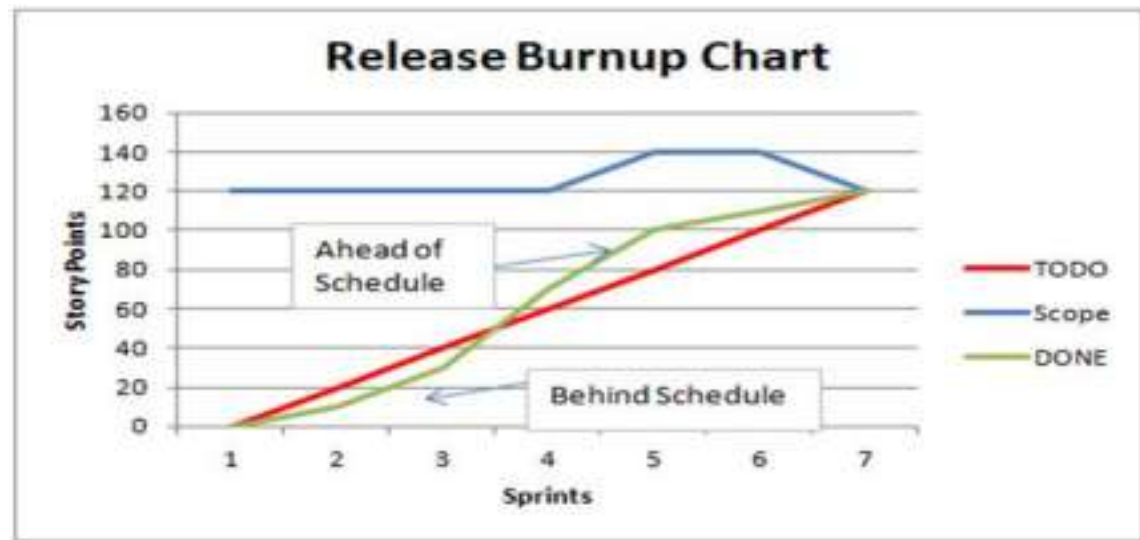
- Goal: Reach zero remaining work by end of sprint.

Burnup Chart:

- Tracks completed work over time **and** total scope.
- Two lines:
 - Work Done (grows upward)
 - Total Scope (may grow with changes)
- Highlights scope creep clearly.

Project XYZ Iteration 1 Burn Down





Ideal vs Actual Lines

- **Ideal Line:**

- Represents perfect progress if work is done evenly.

- **Actual Line:**

- Represents real progress; deviations help spot issues.
- A gap between lines shows over/underestimation, delays, or blockers.

Sprint Burndown vs Release Burndown

| | Type | Sprint Burndown | Release Burndown |
|-----------|------------------------|------------------------|-----------------------------------|
| | | | Multiple sprints (entire release) |
| Focus | Single sprint progress | | |
| Use Case | Daily team sync-ups | | Stakeholder reporting |
| Timeframe | Short (1-4 weeks) | | Medium to long term (2-6 |

months)

When to Prefer Burnup Over Burndown

Prefer Burnup When:

- Scope is changing frequently.
- You want to show how far you've come **and** how much is left.
- Stakeholders need clear visibility on scope creep or adjustments.

Prefer Burndown When:

- Scope is relatively stable.
- You need a simple, fast view of what's left to do.

- Used mostly in team-focused sprint tracking.

Identifying and Mitigating Risk

Risk identification in Agile is not a one-time event (like in traditional waterfall) — it happens **continuously** at every sprint, every meeting, and every conversation.

Agile teams **proactively** look for risks and build **mitigation strategies** early before the risks become issues.

Mitigation strategies include:

- Changing backlog priorities
- Adjusting sprint goals
- Improving team communication

- Enhancing technical practices (like better testing, integration)

Definition and Types of Risk

Risk: A potential problem or opportunity that may impact project outcomes — in either a positive (opportunity) or negative (threat) way.

Types:

- **Technical Risk:** Related to technology challenges (e.g., new frameworks, integration issues)
- **Business Risk:** Misalignment with user needs, market changes
- **Resource Risk:** Team availability, lack of expertise, tools shortage
- **Operational Risk:** Daily workflow problems, miscommunications
- **External Risk:** Outside changes — vendors failing, legal regulations, pandemics

Difference in Risk Handling

Aspect Traditional Projects Agile Projects

When risk is handled Upfront during planning Continuously throughout

Who manages risk Project Manager only Whole team collectively

Risk plan updates Rarely updated formal processes
once finalized Updated sprint by sprint

Response to change Slow due to Fast and reliable

Agile doesn't eliminate risk, it embraces it and reacts faster.

Sources of Risk in Agile Env.

Changing Requirements: As customers understand more, their expectations evolve.

Dependencies: Agile relies heavily on cooperation with other teams, tools, external parties.

Team Skills: Agile needs strong technical skills, team maturity, self-management.

Velocity Fluctuation: Teams may not always deliver the same speed due to complexity or team dynamics.

Risk Identification Methods

Sprint Planning: Discuss risks for selected backlog items

Backlog Grooming: Surface risks while refining backlog items

Daily Standups: Identify small blockers turning into bigger risks

Retrospectives: Reflect and find repeating patterns of risk

Risk Brainstorming: Special session where team discusses:
“What could go wrong?”

Sprint Planning & Backlog Grooming

Sprint Planning:

- Break down backlog items.
- Discuss technical or resource risks for each item.
- Adjust scope if risk is too high.

Backlog Grooming:

- Identify risky stories (too large? too vague? external dependencies?)
- Refine and split stories to lower risk.

Retrospectives and Daily Standups

Retrospective:

- Teams ask:

- What risks materialized last sprint?
- What risks did we manage well?
- What can we change to handle risks better?

Daily Standups:

- If a blocker persists, treat it as a **risk escalation**.
- Action immediately to avoid sprint failure.

Risk Brainstorming

Format: 15–30 minutes session.

Ask questions like:

- "Where do we feel uncertain?"

- "What assumptions are we making?" ●
- "What external factors can affect us?"

Document Risks openly and assign owners.

Agile Practices that Inherently Reduce Risk

Short Iterations and Frequent Delivery

- Frequent delivery gives early feedback.
- Mistakes or misunderstandings are caught quickly.

Continuous Integration and Testing

- Integrating code daily and testing every build finds bugs early.

- Less integration held at the end of projects.

Cross-Functional Teams and Collaboration

- Teams with mixed skills (Dev, Test, UX) can resolve issues faster.
- Reduces bottlenecks when specific skills are needed.

Proactive Risk Response

- Teams adjust backlogs, priorities, or technical approaches before big problems hit.

Prioritizing Risks Using Impact/Probability Matrix

Impact = Severity of risk if it happens.

Probability = Likelihood that the risk will happen.

Critical Priority: Act immediately.

Monitor: Keep an eye; act if worsens.

| Probability → | Low | Medium | High |
|---------------|----------|------------------|-------------------|
| Impact ↓ | | | |
| Low | Ignore | Monitor | Monitor |
| Medium | Monitor | Mitigate | Immediate Action |
| High | Mitigate | Immediate Action | Critical Priority |

Agile Risk Management Cycle

Cycle repeats every sprint:

1. **Identify:** Spot potential risks.
2. **Assess:** Judge likelihood and impact.
3. **Mitigate:** Take steps to reduce risk severity/probability.
4. **Monitor:** Track status during standups and sprint reviews.
5. **Adapt:** Adjust plans, processes, team actions as needed.

Analogy: Think of risk management in Agile as a daily "weather forecast" — we predict, prepare, and adjust.

Responding to Risks Dynamically

Agile allows changing sprint goals mid-sprint if a serious risk appears.

Backlog priorities can be shifted based on newly identified risks.

Teams may even cancel or modify sprint goals if risks threaten sprint success.

Dynamic response is supported through:

- .Quick decision-making (team autonomy)
- .Lightweight documentation (user stories, not heavy specs) .
- Close customer collaboration (instant feedback)

Agile's flexibility is a risk defense mechanism, not a weakness.

In Agile, **risk management is everyone's job, every day.**

Short feedback loops (sprint, review, retrospective)
reduce big disasters.

Dynamic reaction to risk makes Agile robust in complex, changing environments.

alok.giri@ncit.edu.np

What are PM Tools?

Definition: PM tools are digital platforms that help teams plan, execute, monitor, and close projects efficiently.

Purpose: Enhance productivity, visibility, and team

collaboration.

Core Capabilities:

- Task assignment and tracking
- Deadline and milestone planning
- Real-time updates
- Reporting and analytics

Agile Overview

Agile = **iterative**, **incremental**, and **collaborative** approach.

Key practices:

- Sprints

- Daily stand-ups
- Backlogs
- Continuous feedback

Why PM tools? Agile requires structure, visibility, and flexibility—all offered by modern PM tools.

Role of PM Tools in Agile

Sprint Planning: Organize work into defined sprints.

Backlog Management: Prioritize and refine user stories.

Visual Boards: Represent work as tasks/stories visually (Kanban, Scrum boards).

Continuous Feedback Loops: Enable fast tracking of blockers, comments, and changes.

Burndown Charts: Show work completed vs. remaining.

Importance of PM Tools in Collaborative, Distributed, and Iterative Teams

Collaboration in Agile Teams

- Agile thrives on **team interaction**.
- PM tools:
 - Share common boards and dashboards.
 - Enable commenting and real-time updates.
 - Encourage shared ownership of work.

Distributed & Remote Teams

- Increasingly common in global workplaces.
- PM tools support:
 - Async updates across time zones.
 - Cloud-based access and mobile compatibility.
 - Integration with video conferencing and chat (e.g., Zoom, Slack).

Key Features of PM Tools

Core Functionalities

- **Task Creation and Assignment:**
 - Assign users, set due dates, set priorities.
- **Task Hierarchy:**
 - Stories > Tasks > Subtasks (JIRA)

- **Checklists:** Mark off granular steps within tasks.

Time and Sprint Management

- **Sprint Boards:** View current workload.
- **Burndown Charts:** Visualize how much work remains. •

Time Tracking: Log hours, monitor time spent per user/task.

Team Collaboration

- **Mentions:** Alert team members via @mention.
- **Comments & Threads:** Maintain context on task-specific discussions.
- **Activity Logs:** Who did what, when.

Integration Capabilities

- **DevOps:**

- GitHub/GitLab: commit linking, PR tracking
- Jenkins: deployment status
- **Comms & Docs:**
 - Slack, Teams: real-time alerts
 - Google Drive, Dropbox: attach files, collaborate in real time

Dashboards and Reports

- **Monitor:**
 - Sprint velocity
 - Workload by assignee
 - Cycle time, lead time
- **Build:**
 - Custom dashboards for team leads
 - Automated email reports

Best Practices in Using PM Tools

Align Tool with Your Agile Process

- Match workflows to methodology:
 - Scrum? Use Sprint planning + Review + Retrospective templates.
 - Kanban? Focus on WIP limits and cycle times.
- Avoid complexity—only use what's needed.

Maintain Clean Boards

- Daily updates are essential.
- Groom backlogs regularly.
- Define **Done**: criteria should be clear and enforced.

Best Practices in Using PM Tools

Use Communication Features Strategically

- Don't overuse notifications—use @mentions meaningfully.
- Encourage in-tool discussions rather than email.
- Keep decisions recorded in comments or attachments.

Monitor and Reflect

- Analyze reports after every sprint:
 - What slowed us down?
 - Were goals met?
- Feed insights into retrospectives and planning.

Collaboration Tools: Enhancing Team Communication

What Are Collaboration Tools?

- Digital platforms that support communication, file sharing, task coordination, and decision-making.
- Used across teams to:
 - Improve responsiveness
 - Reduce miscommunication
 - Document and track decisions

Categories of Collaboration Tools

| Category | Example Tools | Use Case |
|--------------------|--------------------------|---------------------------------|
| Messaging/Chat | Slack, MS Teams | Quick discussions, updates |
| Video Conferencing | Zoom, Google Meet | Meetings, demos, interviews |
| File Sharing | Google Drive, Dropbox | Document collaboration, storage |
| Project Management | Trello, JIRA, Asana | Task tracking, sprints |
| Documentation | Confluence, Notion, Docs | Knowledge base, policies |
| Whiteboarding | Miro, Mural | Brainstorming, visual ideation |

Modes of Communication

| Type | Examples | Pros | Cons |
|--------------|-----------------------|----------------------------------|-----------------------------------|
| Synchronous | Chat, Calls, Zoom | Real-time, instant clarification | Requires overlapping availability |
| Asynchronous | Email, Docs, Comments | Flexibility, thoughtful replies | Slower feedback loops |

When to Chat, Call or Document

- **Chat:** Quick updates, informal, non-blocking
- **Call/Video:** Urgent or sensitive topics, team alignment
- **Document:** Long-term reference, processes, decisions

Communication in Agile Teams

Agile Communication Needs

- Short feedback cycles (e.g., daily standups)
- Transparent backlog and sprint progress •
- Real-time updates from all team members •
- Continuous improvement via retrospectives

Key Principles for Agile Collaboration

- Visibility: Everyone sees progress & blockers •
- Alignment: Shared goals, clear responsibilities •
- Feedback: Open, frequent, actionable

Common Communication Challenges

Remote Work & Distributed Teams

- Reduced face-to-face cues
- Harder to build trust and rapport
- Need for intentional communication rituals

Time Zones & Availability

- Misaligned working hours → Delays
- Strategies:
 - Use asynchronous updates
 - Shared calendars with working hours
 - Plan meetings inclusively

Cross-Functional Teams

- Different terminologies, priorities
- Need for:
 - Shared documentation

- Common toolset
- Clear communication protocols

Best Practices for Team Communication

- Define preferred channels for each type of message
- Encourage documentation and transparency ●
- Balance between synchronous & async ●
- Regularly review and adapt communication norms

Tool Selection Tips

- Match tools to team size and workflow
- Integrate tools for seamless use
- Prioritize usability & adoption

What is Continuous Integration (CI)?

Definition:

Continuous Integration is the practice of **merging all developers' code changes into a shared mainline** several times a day. Each merge is verified by an automated build and tests.

Key Concepts:

- Frequent commits (daily or more)
- Automated testing and build verification
- Catch issues early in development

Benefits:

- Early bug detection
- Faster feedback
- Reduced integration issues

What is Continuous Deployment (CD)?

Definition:

Continuous Deployment is the practice of **automatically deploying every change that passes all stages of the CI pipeline to production.**

Variants:

- **Continuous Delivery:** Automated staging, manual production deployment
- **Continuous Deployment:** Fully automated, including production

Benefits:

- Shorter release cycles
- Increased deployment frequency
- Faster customer feedback

Importance of CI/CD in Agile

Agile emphasizes **rapid, iterative development**

CI/CD supports **quick feedback loops**

Enhances **collaboration between dev, QA, and ops**

Ensures working software is **always available**

Overview of CI/CD Pipeline Tools

Categories:

- **Version Control Systems**
- **CI Servers / Automation Tools**
- **Build Tools**

- **Testing Tools**
- **Deployment Tools**
- **Monitoring and Notifications**

Version Control Systems

Git:

Type: Distributed Version Control System (DVCS)

Features:

- Every developer has a local copy of the full repository. • Enables branching, merging, rollback, and tracking of code history.
- Fast and efficient collaboration.

Use in CI/CD: Commits and pushes to Git trigger pipeline runs.

GitHub:

Type: Cloud-hosted Git repository

Features:

- Pull requests for code review
- GitHub Actions for CI/CD
- Issue tracking, wikis, and projects

CI/CD Relevance: GitHub Actions offers native automation for builds, tests, and deployments.

GitLab

- **Type:** DevOps platform with Git hosting + built-in CI/CD

- **Features:**

- Git repository + CI/CD + issue tracking + code review in one
 - YAML-based pipeline configuration

- **CI/CD Relevance:** GitLab CI is integrated; triggers jobs on push, merge, etc.

CI Servers/Automation Tools

Jenkins

- **Type:** Open-source CI server
- **Features:**
 - Plugin-based architecture
 - Automates build, test, deploy tasks
 - Highly customizable
- **Use Case:** Flexible for large projects with custom workflows.

GitHub Actions

- **Type:** CI/CD automation directly in GitHub
- **Features:**
 - Triggers on events (push, pull request, etc.)
 - YAML-based workflows
 - Reusable actions from the GitHub Marketplace •
- **Advantage:** Easy setup for teams already using GitHub.

GitLab CI

- **Type:** Integrated CI/CD in GitLab
- **Features:**
 - `.gitlab-ci.yml` to define stages/jobs
 - Runs jobs in Docker containers
- **Advantage:** Unified experience from repo to deployment.

CircleCI

- **Type:** Cloud-native CI/CD platform
- **Features:**
 - Fast build speeds with parallelism
 - Supports Docker, macOS, Linux environments ●
- **Ideal For:** Teams needing fast feedback and scaling.

Build Tools

Maven

- **Type:** Build automation tool for Java
- **Features:**
 - Uses `pom.xml` for configuration
 - Manages dependencies and builds
- **CI/CD Use:** Automatically compiles Java code, runs tests, and creates artifacts.

Gradle

- **Type:** Modern build tool for Java, Android
- **Features:**
 - Faster than Maven (uses a build cache)
 - Uses Groovy or Kotlin for scripting

- **CI/CD Use:** Efficient builds, especially in Android projects.

Npm (Node Package Manager)

Type: Package manager for JavaScript

Features:

- Manages libraries/dependencies
- Defines build/test scripts in `package.json`

CI/CD Use: Installs packages, builds front-end apps, runs tests.

Testing Tools

JUnit

- **Type:** Java unit testing framework
- **Features:**
 - Tests individual classes/methods

- Supports annotations, assertions
- **Use in CI/CD:** Run in CI to catch regressions early.

Selenium

- **Type:** Web browser automation tool
- **Features:**
 - Simulates user interactions in browsers
 - Supports multiple languages
- **Use in CI/CD:** Validates front-end functionality in real browsers.

Cypress

- **Type:** Modern end-to-end testing tool for JavaScript
- **Features:**
 - Fast execution
 - Runs inside the browser
 - Real-time UI during test runs
- **Use in CI/CD:** Ideal for React, Angular, Vue apps.

Postman

- **Type:** API testing tool
- **Features:**
 - Create, send, and validate HTTP requests
 - Automate tests using Newman CLI
- **Use in CI/CD:** Validate API endpoints after every code change.

Deployment Tools

Docker

- **Type:** Containerization platform
- **Features:**
 - Packages app + dependencies into containers
 - Ensures consistency across environments
- **CI/CD Role:** Build Docker images → push to registry → deploy to staging/prod.

Kubernetes


- **Type:** Container orchestration system
- **Features:**
 - Manages clusters of Docker containers
 - Handles scaling, rolling updates, recovery
- **CI/CD Role:** Automate deployment of Docker containers to a scalable cluster.

Monitoring and Notification

Slack

- **Type:** Team communication tool
- **Features:**
 - CI/CD notifications via webhooks or integrations
 - Real-time alerts for build failures, deployments
- **CI/CD Use:** Notifies teams instantly on failures or successes.

Email Alerts

- **Type:** Traditional communication channel
 - **Features:**
 - Sends build/test results to developer inbox
 - Configurable in most CI tools (e.g., Jenkins, GitHub Actions)
 - **CI/CD Use:** Keeps stakeholders informed even if they aren't using Slack.
- 

Code Commit: Developer pushes code to Git repo (e.g., GitHub)

Trigger Build: CI server (e.g., Jenkins) detects the commit **Run**

Tests: Unit, integration, UI, API tests executed **Build Artifacts:**

App is compiled and packaged

Deploy to Staging: App is deployed in a safe test environment

Approval (optional): Manual gate in Continuous Delivery

Deploy to Production: Automated (CD) or Manual **Monitor &**

Notify: Alerts sent to Slack/Email

alok.giri@ncit.edu.np

What is TDD?

Definition:

Test-Driven Development (TDD) is a software development approach where developers **write tests before writing the actual code** that satisfies those tests.

Core Idea:

- Tests drive the design and structure of the software.
- You only write enough code to pass the test.

Key Characteristics:

- Focus on small, iterative development
- High level of test coverage
- Refactoring happens with confidence

TDD vs Traditional Testing

The TDD Cycle – Red → Green → Refactor

Overview:

TDD follows a **three-step cycle** for every feature or function:

1. **Red** – Write a failing test

2. **Green** – Write just enough code to make the test pass 3.

Refactor – Improve the code without changing functionality

Step 1 – RED: Write a Failing Test

Goal:

Start by writing a **test for a specific behavior or requirement**. The test should **fail initially** because the corresponding code doesn't exist yet.

Why?

- Confirms that the test detects missing or incorrect functionality
- Prevents false positives

Example (Python):

python

```
def test_addition():  
  
    assert add(2, 3) == 5 # The function 'add' is not
```

implemented yet

Step 2 – GREEN: Make the Test Pass

Goal:

Write **just enough code** to pass the test. Avoid over engineering or adding unnecessary features.

Focus:

- Minimal implementation
- Avoid premature optimization

Example (Python):

```
python
def add(a, b):
    return a + b
```

Important: Don't worry about elegance or edge cases at this point. Just

make the test pass.

Step 3 – REFACTOR: Improve the Code

Goal:

Clean and improve the implementation without breaking any tests.

Common Refactorings:

- Rename variables
- Simplify logic
- Remove duplication
- Apply design patterns

Safety Net:

All tests remain **green**, ensuring that behavior hasn't changed.

Benefits of TDD

1. Higher Code Quality:

- Forces developers to consider use cases and edge cases early.

2. Less Debugging:

- Issues are caught immediately during development.

3. Better Design:

- Encourages modular, loosely-coupled code.

4. Improved Developer Confidence:

- Easy to refactor and extend code safely.

5. Living Documentation:

- Tests describe what the code is supposed to do.

6. Faster Feedback Loop:

- Errors are detected quickly and fixed early.

Challenges of TDD

1. Initial Learning Curve:

- Requires change in mindset and discipline.

2. Slower Start:

- Initial development may feel slower due to writing tests first.

3. Over-Testing:

- Risk of writing tests for trivial code.

4. Refactoring Tests:

- Test maintenance can be time-consuming if code changes frequently.

5. Difficult for UI/UX Logic:

- Not all layers (like UI/UX) are easily testable in TDD fashion.

When to Use and Avoid TDD

Use TDD:

- Backend services
- Algorithms and core business logic
- APIs and SDKs
- Applications requiring long-term maintenance

Avoid TDD:

- Rapid prototyping
- UI/UX-heavy design where visual feedback is key
- One-off scripts or disposable code

What is BDD?

Definition:

Behavior-Driven Development (BDD) is a software development practice that encourages **collaboration between developers, QA, and non-technical stakeholders** to define the behavior of a system in **plain language**.

Core Idea:

- Uses **natural language constructs** to describe software behavior
- Focuses on the **expected outcome** rather than implementation

Key Principle:

If we can describe it clearly, we can build it correctly.

Evolution from TDD to BDD

BDD vs TDD

| Criteria | TDD | BDD |
|---------------|------------------------------|---|
| Goal | Validate functionality | Validate behavior |
| Test Format | Code-based assertions | Readable scenarios (Given-When-Then) |
| Communication | Developer-centric | Cross-team (dev, QA, business) |
| Documentation | Often buried in code | Living documentation |
| Readability | Requires technical knowledge | Designed for non-technical stakeholders |

Structure of BDD – Given, When, Then

The Gherkin Syntax:

- **Given:** Initial context or precondition
- **When:** Event or action taken
- **Then:** Expected outcome or result

Advantages:

- Makes requirements **testable**
- Easy for all stakeholders to **understand and verify**

Writing Effective BDD Scenarios Tips

for Clear, Concise, and Testable Scenarios:

1. Focus on business value:

- Scenarios should reflect real user behavior.

2. Use simple and consistent language: ○ Avoid technical jargon.

3. Keep scenarios short and atomic:

- One behavior per scenario.

4. Avoid overlapping conditions:

- Separate out unrelated behaviors.

5. Use roles and actions clearly:

- Who is doing what and why?

Benefits of BDD

1. Enhanced collaboration:

- Aligns technical and non-technical stakeholders.

2. Improved clarity of requirements:

- Business rules are explicitly defined in scenarios.

3. Living documentation:

- Scenarios serve as executable specs and up-to-date documentation.

4. Reduced miscommunication:

- Shared understanding through common language.

5. Faster test creation:

- Reusable steps and readable formats make test writing easier.

Challenges of BDD

1. Initial setup and training:

- Requires learning tools and process changes.

2. Overhead for simple projects:

- May be too heavy for small scripts or prototypes.

3. Scenarios can become too detailed:

- Risk of becoming verbose and brittle if over-specified.

4. Maintenance of steps:

- Common step definitions must be well-organized.

5. Misuse as a testing tool only:

- BDD is about collaboration and behavior, not just testing.

When to Use BDD

Best suited for:

- Agile teams with frequent collaboration
- Feature-rich applications with evolving requirements
- Projects where behavior is key to success (e.g., user workflows)

Avoid or limit BDD when:

- Team lacks communication with business users
- Small, one-off tools or utilities

- When requirements are too vague or volatile to formalize

Overview of Collaborative Programming

Definition:

Collaborative programming involves **two or more developers working together** in real-time on the same code base.

Why it matters:

- Encourages **continuous feedback**
- Promotes **collective ownership**
- Enhances **code quality and team learning**

Forms:

- **Pair Programming:** 2 developers
 - **Mob Programming:** 3 or more developers (often the whole team)

What is Pair Programming?

Definition:

Pair programming is a development technique where **two developers work together at one workstation** on the same task.

- One developer types the code (**Driver**)
- The other reviews and guides (**Navigator**)
- They frequently **switch roles**

Quote:

“Pair programming is a dialog between two people trying to simultaneously program and understand the problem and its solution.” – Laurie Williams

Roles in Pair Programming

1. Driver

- Writes the code
- Focuses on **syntax and implementation**
- Pays attention to immediate tasks

2. Navigator

- Reviews each line as it's written
 - Thinks about **design, strategy, and direction** ●
- Spots potential bugs, edge cases, and improvements

Modes of Pair Programming

1. Expert–Novice

- Experienced developer guides a beginner
- Great for **mentorship and onboarding**
- Balance required to keep the novice engaged

2. Ping-Pong Pairing

- One writes a **failing test**, the other writes **code to pass it**
- Then switch roles
- Follows **TDD** style

3. Remote Pairing

- Pairs work remotely using tools like **VSCode Live Share**, **Tuple**, **CodeTogether**
- Requires clear audio, fast internet, and screen-sharing

Benefits of Pair Programming

1. Improved code quality:

- Two sets of eyes reduce bugs and increase clarity

2. Faster knowledge sharing:

- Developers learn from each other

3. Better design decisions:

- Real-time discussion encourages thoughtful solutions

4. Increased team cohesion:

- Builds communication and mutual trust

5. Reduced bottlenecks:

- No single point of failure or dependency on one person

Challenges of Pair Programming

1. Initial drop in productivity:

- Can feel slower until the team adjusts

2. Personality mismatches:

- Requires interpersonal skills and mutual respect

3. **Fatigue:**

- Pairing can be mentally intense without breaks

4. **Not all tasks are suitable:**

- Trivial or repetitive tasks may not benefit

5. **Scheduling difficulties:**

- Matching availability can be tough in distributed teams

What is Mob Programming?

Definition:

Mob programming is a style of programming where **the whole team works together on the same task, at the same time, on the same computer.**

- One person is the **Driver**
- Everyone else acts as **Navigators**

- All decisions are made **collaboratively**

Quote:

“All the brilliant minds working on the same thing, at the same time, in the same space, and at the same computer.” – Woody Zuill

Roles in Mob Programming

1. Driver

- The only person typing
- Implements what is discussed
- **Does not make decisions alone**

2. Navigators

- Everyone else

- Discuss and guide the Driver
- Suggest improvements, spot issues, and strategize

Rotation Tip: Rotate the Driver every 10–15 minutes

Remote and In-Person Mob Programming

In-Person:

- One workstation with shared screen
- Use timer for rotation

Remote:

- Use tools like:
 - **Zoom/Teams/Google Meet** for voice/video
 - **VSCode Live Share**
 - **Miro or digital whiteboards** for brainstorming

Best Practices:

- Strong facilitation
- Clear goals and structure
- Frequent short breaks

Benefits of Mob Programming

1. Rapid knowledge sharing:

- Everyone learns together

2. Higher code quality:

- Multiple reviewers catch issues early

3. Shared ownership:

- Everyone understands the code base

4. Fewer interruptions:

- Team is aligned and focused

5. Real-time mentoring:

- Juniors learn from seniors instantly

Challenges of Mob Programming

1. Logistics and time zones:

- Harder to schedule full-team sessions

2. Overcommunication:

- Requires structure to avoid chaos

3. Burnout risk:

- Intense focus for extended periods

4. Cost concerns:

- Appears expensive (entire team working on one thing)

5. Requires strong facilitation:

- Otherwise can become inefficient

When to Use Pair vs Mob Programming

Use Pair Programming When:

- Two developers can work efficiently on a story
- Mentoring or onboarding a teammate
- Refactoring or debugging a specific feature

Use Mob Programming When:

- Complex or high-risk feature needs many inputs
- Onboarding a new team or aligning understanding
- Architectural decisions or major codebase changes

Why Focus on Maintainability?

Definition of Maintainability:

- The ease with which a software system can be understood, changed, and extended.

Why It Matters:

- Code is read far more often than it is written.
- Long-term cost of software is mostly **maintenance**, not initial development.

What is Refactoring?

Definition:

Refactoring is the process of **improving the internal structure of code** without changing its external behavior.

Key Goals:

- Enhance readability
- Improve design
- Reduce complexity
- Remove code smells

"Refactoring is like cleaning up your workspace—nothing changes in function, but everything becomes easier to work with." – Martin Fowler

Technique

Purpose

Rename Variable/Method

Improve clarity and naming

Extract Method

Break down long functions

Inline Method/Variable

Remove unnecessary abstraction

Remove Dead Code

Eliminate unused or unreachable code

Replace Magic Numbers

Use named constants for readability

Encapsulate Fields

Use getters/setters to protect state

Simplify Conditionals

Use guard clauses, remove nesting

Split Large Classes

Follow SRP and reduce coupling



Refactoring Techniques

When Should You Refactor?

Best Times to Refactor:

- **Before adding a new feature:** Clean up the area you'll work on.
- **While fixing bugs:** Understand and improve the faulty area. ●
- After code reviews:** Address quality feedback.
- **During TDD cycles:** The "Refactor" stage in Red-Green-Refactor.

Avoid:

- Refactoring without tests or clear understanding ●
- Big bang refactors with no incremental checkpoints

Benefits of Refactoring

1. Improved readability and understanding
2. Lower technical debt
3. Easier debugging and modification
4. Improved performance (when optimizing)
5. Better team collaboration through clean, consistent code

What is a Code Review?

Definition:

A code review is a **systematic examination of source code** by peers to identify bugs, improve code quality, and share knowledge.

Purpose:

- Find issues **before they reach production**
- Encourage **best practices**
- Promote **team-wide standards**

Types of Code Reviews

1. Synchronous Reviews:

- Real-time discussion via **pairing or walkthrough meetings**
- Tools: Screen sharing, IDE collaboration (e.g., Live Share)

2. Asynchronous Reviews:

- Developer submits a pull request (PR); reviewers comment later
- Common in distributed teams

- Tools: GitHub, GitLab, Bitbucket, Phabricator
- ## Reviewer Mindset

Constructive:

- Provide **specific and actionable** feedback.
- Avoid sarcasm or nitpicking.



Respectful:

- Use inclusive and polite language.
- Focus on the code, **not the coder**.

Collaborative:

- Ask questions instead of making assumptions.
- Offer suggestions, not demands.

Examples:

-  “This code is awful.”
-  “Could we simplify this method for readability?”

Author Mindset

Open to Feedback:

- Assume good intent from reviewers.
- Be willing to learn and adapt.

Prepare for Review:

- Run all tests
- Write meaningful commit messages
- Add comments for complex logic

Communicate Clearly:

- Tag reviewers appropriately • Explain

reasoning behind tricky code

Benefits of Code Reviews

- 1. Early bug detection**
- 2. Improved code quality**
- 3. Knowledge sharing among team members**
- 4. Mentorship and skill development**
- 5.**

Consistent coding standards

- 6. Team accountability and cohesion**

Refactoring + Code Review = Healthy Codebase

Why They Work Together:

- Refactoring improves code quality **internally**
- Code reviews enforce quality **externally**
- Together, they ensure the system remains clean, scalable, and

robust

Best Practice:

- Refactor **before submitting** code for review
- Use review comments as **refactoring triggers**

alok.giri@ncit.edu.np

What is SAFe?

Definition:

SAFe (Scaled Agile Framework) is a **set of organizational and workflow patterns** intended to help enterprises **scale agile**

practices across teams, business units, and portfolios.

Purpose:

- Align **strategy with execution**
- Foster **collaboration across large teams**
- Deliver **value at scale**

Origin and Evolution of SAFe

- **Created by:** Dean Leffingwell
- **First published:** 2011
- Combines principles from:
 - **Agile**
 - **Lean**
 - **Systems thinking**
 - **DevOps**

- Has evolved through multiple versions; latest version (as of 2024): **SAFe 6.0**

Why it matters:

- Designed for **large-scale coordination** across dozens or even hundreds of agile teams

When Should You Consider SAFe?

SAFe is suitable when:

- You have **multiple agile teams** working on interdependent solutions
- You need **alignment between business strategy and technology delivery**
- There's a need for **predictable delivery of value**

- You're in a **regulated or complex industry**

SAFe is not ideal for:

- Small teams or startups
- Organizations not ready for structured change

Introduction to ART (Agile Release Train)

Definition:

An Agile Release Train (ART) is a **long-lived team of agile teams** (typically 50–125 people) that plan, commit, and deliver **together**.

Key Concepts:

- Operates on a **fixed cadence (Program Increment – PI)**, typically 8–12 weeks
- Aligns team objectives with business priorities

- ARTs include all roles required for delivery: Dev, QA, UX, Product, and more

Goal:

Deliver **continuous value** across teams and stakeholders

SAFe Configurations Overview

SAFe is **configurable** to match organizational complexity:

1. Essential SAFe

- The most basic and **foundational** configuration
- Includes: Agile Teams, ART, PI Planning, ScrumXP, Kanban

2. Portfolio SAFe

- Adds strategy and funding alignment

- Includes: Lean Portfolio Management (LPM), Strategic Themes

3. Large Solution SAFe

- Supports **large-scale systems** that require multiple ARTs
- Includes: Solution Trains, Solution Architect, Solution Management

4. Full SAFe

- Combines all configurations
- Used in **complex, enterprise-scale** organizations

Four SAFe Configurations

- Essential SAFe: Base level
- Portfolio SAFe: Adds strategic alignment
- Large

Solution SAFe: Adds solution-level coordination • Full

SAFe: Combines all above for enterprise-level agility

Key Roles in SAFe

| Role | Primary Responsibility |
|------------------------|---|
| Release Train Engineer | Facilitates ART operations |
| Product Management | Defines and prioritizes Features across teams |
| System Architect | Guides technical vision and enablers |
| Agile Teams | Cross-functional dev/test/UX teams |
| Scrum Master | Facilitates team-level agile process |
| Product Owner (PO) | Owns team backlog, defines stories |

Putting It All Together – SAFe in Action

Cycle Example:

1. **PI Planning:** Teams commit to objectives
2. **Iterations:** Agile Teams deliver incrementally
3. **System Demo:** Working software shown at end of each iteration
4. **Inspect & Adapt:** Teams reflect, improve, and re-align

Collaboration Across Levels:

- Portfolio → ART → Agile Teams
- Strategy → Execution → Feedback

Benefits of SAFe

1. Alignment between business and IT
2. Predictable, frequent delivery of value
3. Transparency and visibility across teams
- 4.

Improved product quality

5. Cross-team collaboration and shared responsibility

Challenges of SAFe

1. Heavyweight process for smaller teams
2. Requires cultural change and buy-in
3. Complex role structures and terminology
4. Training and onboarding costs
5. Can become bureaucratic if misapplied

When is an Agile Team Considered ‘Large’?

Typical Agile Team Size:

- 5 to 9 members is ideal (as per Scrum Guide)

‘Large’ in Agile Context:

- More than **2-3 agile teams** working on the same product or platform
- More than **10–12 people** trying to coordinate work on related outcomes

Challenges Begin When:

- Teams need **shared coordination**
- Dependencies increase across teams
- **Communication overhead** becomes non-trivial

Why Scaling Agile Is Hard

- Agile thrives on **collaboration and quick feedback**

- Scaling brings risks of:
 - **Misalignment**
 - **Duplicated efforts**
 - **Inconsistent velocity**
 - **Slow decision-making**
- Requires structure **without sacrificing agility**

Coordination Strategies

1. Sync Through Events

- Schedule recurring cross-team meetings:
 - Program Increment (PI) Planning
 - Release Demos

- Inspect and Adapt (I&A) workshops

2. Scrum of Scrums

- Representatives from each team meet regularly
- Share progress, blockers, and dependencies ●

Helps with **horizontal alignment**

Coordination Strategies

1. Sync Through Events

- Schedule recurring cross-team meetings:
 - Program Increment (PI) Planning
 - Release Demos
 - Inspect and Adapt (I&A) workshops

2. Scrum of Scrums

- Representatives from each team meet regularly •
- Share progress, blockers, and dependencies
- Helps with **horizontal alignment**

3. Product Owners Sync

- Aligns backlog priorities across teams
- Ensures **product strategy coherence**
- Supports coordinated decision-making on scope tradeoffs

PI Planning (in SAFe)

Definition:

A large-scale planning event (2-day) where all teams on an ART align on:

- What they'll deliver in the upcoming Program Increment (8–12 weeks)
- Risks and dependencies

PI Planning Activities:

- Review vision and roadmap
- Draft team plans
- Identify cross-team dependencies
- Commit to objectives

Benefits:

- Alignment, transparency, shared ownership

Shared Goals and Metrics

Why Shared Goals Matter:

- Prevent isolated efforts
- Foster collaboration
- Prioritize value delivery

Shared Metrics:

- Velocity trends across teams
- Feature and story cycle time •
- Deployment frequency
- Defect escape rate
- Objective completion % per PI

Integrating Multiple Agile Teams

Integration is Key for:

- System-level testing
- Avoiding integration hell
- Continuous delivery

Strategies:

- **Define clear interfaces/APIs**
- **Test integration early and often** ●

Use shared staging environments

Horizontal and Vertical Team Slicing

Horizontal Slicing (by layers):

- One team owns UI, another owns backend, another owns DB

Issues:

- Tight coupling
- Dependency hell

Vertical Slicing (by features):

- Teams work end-to-end on features
- Promotes autonomy and faster delivery

Preferred approach in Agile scaling

Summary – Key Takeaways

- Large Agile teams need **intentional coordination and communication**
- Events like **Scrum of Scrums, PO Syncs, PI Planning** provide structure
- **Shared goals** and metrics drive alignment
- Prefer **vertical slicing** for team structure
- Integration should be **continuous and automated**
- The goal is **scaling without losing agility**

alok.giri@ncit.edu.np

Agile KPIs

What are KPIs in Agile?

- **Metrics** that help teams measure progress, efficiency, and predictability
- Track **delivery performance** and highlight opportunities for

improvement

Common Agile KPIs:

- Velocity
- Cycle Time
- Lead Time
- Throughput
- Defect Rate
- Team Happiness (qualitative)

Velocity

Velocity is the amount of work a team completes in a sprint, typically measured in story points or work items.

How to Calculate Velocity

Step-by-step:

1. At the end of each sprint, **sum the story points** for all completed stories.
2. **Do not count** incomplete stories.
3. Track the velocity **across multiple sprints** to find a stable average.

Example:

- Sprint 1: 23 story points
- Sprint 2: 26 story points
- Sprint 3: 25 story points

Average Velocity = $(23 + 26 + 25) / 3 = 24.7 \approx 25$ points

Velocity – Capacity Planning and Forecasting

- **Plan future sprints:** Use average velocity to forecast how much work can fit
- **Estimate delivery dates:** If backlog has 100 story points and velocity is 25 → 4 sprints
- Helps balance **team workload** and **avoid over-committing**

Caution:

Velocity is a **team-specific internal metric**—not meant for comparing teams.

Slide 6: Cycle Time – Definition

Definition:

Cycle Time is the **total time it takes to complete a work item**, from the moment it starts (in progress) to when it's done.

It answers:

“How long does it take to complete a task once we start it?”

How to Calculate Cycle Time

Formula:

Cycle Time = Completed Date – Start Date (in-progress)

Example:

- A story started on May 1 and completed on May 4 → Cycle Time = 3

days

Tools That Help:

- Jira, Azure DevOps, Trello with plugins
- Cumulative Flow Diagrams (CFDs)

Strategies to Reduce Cycle Time

1. Limit Work In Progress (WIP):

- Fewer tasks = more focus and faster flow

2. Reduce bottlenecks:

- Identify stages where work piles up

3. Improve team collaboration:

- Shared ownership and pairing

4. Break down large stories:

- Smaller stories move through faster

5. Automate testing and deployment:

- Reduce delays caused by manual QA or release gates

Lead Time – Definition

Definition:

Lead Time is the **total time from when a request is made** (backlog entry) **to when it is completed**.

It answers:

“How long does a customer wait to get what they asked for?”

Slide 10: How to Calculate Lead Time

Formula:

Lead Time = Completion Date – Request Date

Example: A feature is requested on April 1, and delivered on April 10 → Lead Time = 9

days **Includes:**

- Waiting time before development

- Cycle Time (development phase)

Improving Lead Time

1. Prioritize backlog frequently:

- Keep high-value work near the top

2. Visualize work queues:

- Identify aging items and dependencies

3. Reduce hand-offs and approvals:

○ Empower teams to move work forward faster

4. Manage external blockers:

- Communicate with outside teams early

5. Automate workflows:

- Integrate CI/CD to cut delivery delays

Using All Three KPIs Together

| KPI | Use For |
|------------|--|
| Velocity | Sprint planning and delivery forecasting |
| Cycle Time | Team efficiency and flow improvement |
| Lead Time | Customer satisfaction and delivery speed |

Introduction to Reporting Tools and Dashboards

Why Visualize Agile Metrics?

- Provide **real-time insights** into progress and issues
- Drive **data-informed decisions**
- Improve **transparency and accountability**

- Help in **retrospective analysis** and **future planning**

Goal:

Track how work flows, identify bottlenecks, and assess delivery performance.

Jira Dashboards

Overview:

Jira offers **custom dashboards** with real-time gadgets for:

- Velocity
- Burndown
- Sprint Health
- Cumulative Flow Diagrams
- Control Charts (Cycle Time)

Features:

- Filterable widgets using JQL
- Team-level and program-level dashboards
- Integration with Confluence, Bitbucket, GitHub

Tip: Use “**Filters + Gadgets**” to create focused dashboards per team or role.

Trello and Asana Dashboards

Trello

- Simple Kanban boards
- **Power-Ups** enable reporting (Charts by Vizydrop, Dashcards)
- Great for small teams and simple projects

Asana

- Timeline and Portfolio views
- Dashboards for:
 - Task status
 - Due date tracking
 - Goal progress
- Suited for cross-functional and non-engineering teams

Key Metrics to Visualize

| Metric | Purpose |
|---------------------|---|
| Velocity Chart | Forecast capacity based on past delivery |
| Burndown Chart | Track sprint progress and completion rate |
| Cycle Time | Measure efficiency from start to finish |
| Lead Time | Total time from request to delivery |
| WIP Limits | Prevent overload by capping active work |
| Epic/Release Burnup | Monitor value delivery toward strategic goals |

Velocity Chart

- Shows the number of **story points** (or tasks) completed over sprints ●

Helps with:

- **Capacity planning**
- Sprint forecasting
- Tracking **team consistency**

Jira Tip: Use the **Velocity Report** to analyze across multiple sprints

Slide 8: Burndown Chart

- Plots **work remaining vs time**
- X-axis = sprint days, Y-axis = story points
- **Ideal line vs actual progress**
- Reveals:
 - Blockers
 - Scope creep
 - Overcommitment

Lead Time & Cycle Time Charts

- **Cycle Time:** From “in progress” to “done”
- **Lead Time:** From backlog to done

Visual Tools:

- **Control Charts** in Jira
- **Cumulative Flow Diagrams (CFD)** for identifying WIP congestion

Use cases:

- Optimize flow efficiency
- Reduce delays in development or review

Work in Progress (WIP) Limits

- Shows how many items are currently active • **Limiting**

WIP = Faster flow + less context switching

Metrics to track:

- Average WIP count
- Items per column (e.g., Testing, Code Review)
- Age of WIP items

Tools: Kanban boards with **column limits** and **WIP alerts**

Epic and Release Burnup Charts

Epic Burnup Chart:

- Tracks how much work is completed toward an **epic** goal

Release Burnup Chart:

- Tracks progress toward a **product release**

Benefits:

- Shows **scope changes and additions**
- Provides a **clear picture of delivery pace** vs scope growth

Best in: Jira Advanced Roadmaps, Aha!, or Azure Boards

Best Practices for Dashboards

1. Keep it simple and focused

- Tailor dashboards to team roles (Dev, PO, Management)

2. Automate where possible

- Use real-time data; avoid manual entry

3. Visualize trends, not just snapshots

- Compare sprint-over-sprint, epic trends, etc.

4. Include actionable insights

- Use charts that trigger retrospectives or improvements

5. Review regularly

- Dashboards are only useful if **reviewed and acted upon**

Metrics and Continuous Improvement – The Connection

What is Continuous Improvement?

A systematic effort to seek and implement **incremental changes** that enhance quality, efficiency, and performance.

Role of Metrics:

- Provide **quantitative insight** into how a process performs
- Allow **data-driven decisions** instead of assumptions
- Highlight areas of waste, delays, or inconsistency
- Enable **experiments, validation, and learning**

Agile Principle #12:

“At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.”

Agile Metrics for Continuous Improvement

| Metric | Purpose |
|-----------------|---|
| Velocity | Understand team delivery capacity |
| Cycle Time | Optimize how long it takes to complete work |
| Lead Time | Improve customer satisfaction through faster delivery |
| Escaped Defects | Track quality of delivery |
| Throughput | Analyze how many items are completed over time |
| Cumulative Flow | Visualize bottlenecks and flow efficiency |
| Team Happiness | Assess morale and well-being |

Using Metrics for Process Enhancement

Steps to Leverage Metrics Effectively:

1. Collect Reliable Data:

- Automate tracking via tools (e.g., Jira, GitLab, Azure Boards)

2. Interpret Trends:

- Look for patterns over time, not isolated data points

3. Correlate with Behavior:

- Identify what practices drive improvements (or regressions)

4. Adjust Processes:

- Tweak WIP limits, sprint length, review practices, etc.

5. Validate Results:

- Remeasure after changes

Root Cause Analysis (RCA)

Purpose:

To go beyond symptoms and identify the **underlying cause** of inefficiencies or issues.

Techniques:

- **5 Whys Method:**

- Ask “why?” repeatedly until the core cause is uncovered

- **Fishbone (Ishikawa) Diagram:**

- Categorize causes into people, process, tools, etc.

Example (5 Whys):

- Defect found in production

- Why? Missed in testing
- Why? No test case for edge case
- Why? Requirements unclear
- Why? Inadequate PO review
- Why? Lack of stakeholder collaboration

Improvement Cycles – PDCA Framework

1. Plan:

- Identify an area for improvement using metrics •
- Set a goal or hypothesis for change

2. Do:

- Implement a small, testable change •
- Apply in a sprint or workflow

3. Check:

- Review data after implementation
- Validate with team feedback and metrics

4. Act:

- Standardize if successful
- Adjust and retry if not

SMART Goals for Continuous Improvement

alok.giri@ncit.edu.np

Why Study Case Studies?

- Theoretical Agile \neq Real-World Agile
- Case studies help us:
 - Understand **what works in practice**
 - Learn from **mistakes and challenges**
 - Apply patterns to our own teams

SUCCESS CASE – SPOTIFY

Slide 3: Spotify – Overview

- Founded in **2006**, now a global streaming leader
 - Faced challenge: How to **scale Agile** without losing **speed and autonomy**

Spotify's Motto:

“Think it. Build it. Ship it. Tweak it.”

Spotify Agile Model

Spotify didn't **invent Agile**, but evolved a unique **structure** to scale it:

1. Squads

- Basic unit (~6–12 people), similar to a **Scrum team**
- Cross-functional (Dev, QA, UX, etc.)

- Fully autonomous: owns a feature or product area
- Uses Scrum or Kanban (team decides)
- Has a **Product Owner**

“A squad is like a mini-startup, responsible for a specific aspect of the product.”

Spotify – Tribes, Chapters, and Guilds

2. Tribes

- A **collection of squads** (up to ~100 people)
- Squads in a tribe work on related areas (e.g., playlist, mobile)
- Led by a **Tribe Leader**
- Promotes alignment and cross-squad learning

3. Chapters

- Horizontal group of people with similar skills (e.g., Frontend Devs)
- Belong to **different squads** but the **same tribe**
- Led by a **Chapter Lead** (like a line manager)
- Focus: Tech consistency, skill growth

Guilds

- Informal communities across **tribes and squads**
- Focused on shared interest or practices (e.g., testing, DevOps, design)
- Anyone can join; helps with **knowledge sharing**

FAILURE CASE – NOKIA

Nokia's Agile Attempt

- Dominant phone brand in early 2000s
- Introduced Agile (Scrum) ~2007–2009
- **Intent:** Respond faster to smartphone disruption

- **Reality:** Agile implementation failed to save them

FAILURE CASE – AIRBUS A350

Slide 9: Airbus A350 Agile Integration

- Applied Agile in developing cockpit software
- Teams were **distributed across Europe**
- Some teams followed Agile; others used **Waterfall** or traditional systems engineering

Why did Nokia Fail?

| Root Cause | Details |
|------------------------|---|
| Agile in name only | Adopted ceremonies, not mindset |
| No cultural change | Hierarchical management resisted team autonomy |
| Delivery ≠ Product Fit | Agile speed didn't help bad strategic decisions (e.g., Symbian) |
| Lack of feedback loops | Market signals ignored; no real product feedback |
| Blame culture | Fear stifled innovation and ownership |

“We had Scrum, but not agility.”

Why was Airbus slow?

| Issue | Consequence |
|---------------------------------|---|
| Lack of systemic agility | Agile in isolation = misaligned deliveries |
| Delayed integration | Found mismatches in late-stage testing |
| Geographical and cultural silos | Poor collaboration and communication |
| No full value stream ownership | Agile teams couldn't see impact on full sys |

Result: Costly delays, rework, and partial Agile rollback

PATTERNS AND INSIGHTS

Patterns of Agile Success

1. Teams own what they build
2. Cultural fit with Agile mindset
3. Fast feedback and experimentation
4. Strong leadership support and trust
5. Continuous improvement baked in

Patterns of Agile Failure

1. “Agile theater” – rituals without real change
2. No stakeholder alignment
3. Blame and control-based culture
4. Tech debt ignored
5. Teams constrained by top-down decisions

The Cultural Foundation

Positive Culture

Autonomy & trust

Psychological safety

Learning mindset

Transparency & collaboration

Toxic Culture

Micromanagement

Blame-shaming

Fear of mistakes

Siloed thinking

Spotify's success was cultural before structural.

Continuous Feedback Loops

Why Feedback is Crucial:

- Enables **fast correction**
- Drives **product relevance**
- Empowers teams to **reflect and improve**

Feedback in Agile Layers:

- **Code level:** Pull requests, pair programming
- **Team level:** Retrospectives
- **Customer level:** Reviews, usability testing
- **System level:** Product analytics

What is Agile Transformation?

Definition:

Agile Transformation is the **fundamental change** in how an organization **thinks, delivers value, collaborates, and operates**, by adopting **Agile principles and frameworks**.

It's not just about:

- Doing daily stand-ups
- Installing Jira
- Running sprints

It is about:

- **Mindset shift**
- Structural change
- New ways of **leading, working, and learning**

Why Organizations Choose Agile Transformation

- Faster response to market changes

- Higher customer satisfaction
- Improved team morale and engagement
- Increased visibility and predictability
- Better alignment between business and IT

Key Steps in Agile Transformation

Step 1 – Create Urgency & Executive Buy-in

- Make a strong **business case for change**
- Highlight:
 - Customer dissatisfaction
 - Innovation lag
 - Delivery bottlenecks
- **Involve leadership** early
- Secure executive sponsors to champion the change

Quote:

"Without leadership support, Agile becomes theater."

Step 2 – Define the Vision and Agile Goals

- What does **Agile success** look like for this organization?
- Define outcomes: Faster releases, better quality, more engaged teams

Step 3- Assess Current State (As-Is Analysis)

- Evaluate:
 - Team structures
 - Delivery processes (e.g., Waterfall, ad hoc)
 - Culture and decision-making
 - Tooling and technical practices

Tools:

- Agile Maturity Assessment
- Value Stream Mapping

- Surveys and interviews

Step 4- Design the Target State

- Choose transformation models:
 - **Team-level:** Scrum, Kanban
 - **Scaling frameworks:** SAFe, LeSS, Spotify, Nexus

Design considerations:

- Agile team structures
- Role realignment (PO, SM, dev teams)
- Governance and metrics

Step 5- Form Agile Teams & Define Roles

- Form **cross-functional teams** with clear ownership
- Assign:
 - **Product Owner**

- **Scrum Master or Agile Coach**
- Developers, Testers, Designers

Tip: Avoid partial teams or role confusion. Full-time commitment yields better

agility. **Step 6- Start with Pilots (Proof of Concept)**

- Select **2–3 pilot teams**
- Choose manageable scope
- Give them training, tools, and support

Why Pilots?

- Quick wins
- Learn what works (and doesn't)
- Build momentum with real outcomes

Step 7 – Scale What Works

- Use successful pilots to define **playbooks** ●

Roll out across more teams and business units ●

Support with:

- Agile Coaches
- Communities of Practice
- On-demand training

Scaling Options:

- SAFe for structured orgs
- Spotify model for innovative orgs
- LeSS for product-centric orgs

Step 8 – Measure, Inspect, and Adapt

- Track transformation KPIs:
 - Cycle Time, Lead Time, Team Velocity
 - Employee Engagement

- Customer Feedback
- Conduct **transformation retrospectives**
- Adjust based on feedback and metrics

Continuous transformation > One-time switch



| From | Form of Resistance |
|-------------------|------------------------------------|
| Senior Management | Fear of losing control |
| Middle Management | Threat to authority |
| Teams | Comfort with known processes |
| Support Functions | "This doesn't apply to us" mindset |

Challenge 2 – Misunderstanding Agile

Symptoms:

- Teams “doing Scrum” but not being Agile

- Overfocus on tools and ceremonies
- Agile becomes “delivery speed only” game

Solution:

- Emphasize **Agile values & principles**
- Prioritize **mindset over mechanics**

Challenge 3 – Lack of Role Clarity

- Confusion between **Project Manager** and **Scrum Master** •
- POs expected to write specs instead of manage product vision •
- SMs becoming status reporters instead of coaches

Fixes:- Clear role definitions, Role-based training, Leadership modeling correct behavior

Challenge 4 – Silos and Poor Collaboration

- Agile thrives on **collaboration**

- Silos block:
 - Shared ownership
 - Flow of value
 - Feedback loops

Antidotes:

- Cross-functional teams
- Shared KPIs
- Frequent cross-team planning (e.g., PI Planning)

Challenge 5 – Inconsistent Leadership Behavior

- Saying “Be Agile” while demanding fixed scope/deadlines
- Micromanagement or ignoring Agile values

Solution:

- Train and coach leaders
- Use **Agile Leadership frameworks** (e.g., SAFe LPM, Lean Thinking)

Team Formation and Role Assignment

Agile Planning – Sprint 0

Activities in Sprint 0:

- Understand the **project objective**
- Break down **features into user stories**
- Prioritize the backlog (use MoSCoW or story mapping)
- Estimate effort (e.g., story points)
- Create the **Sprint Plan** (1-week or 2-week iterations)

Tools: Trello, Jira, Notion, physical boards (if in-person)

Iterative Development and Delivery

During Each Sprint:

- **Daily Standups:** Brief updates on progress, blockers
- **Task board updates:** Move cards across Kanban states
- **Sprint Review:** Demo to stakeholders/instructor
- **Sprint Retrospective:** Reflect on process and teamwork

Deliverable Format:

- Working prototype, UI mockup, MVP demo, or service simulation

Continuous Feedback and Adjustments

Internal Feedback:

- Use retrospectives to improve collaboration and delivery
- Adjust WIP, sprint size, or roles as needed

External Feedback:

- Instructor or stakeholder reviews during Sprint Reviews
- Respond with story re-prioritization or change planning

Reminder: Agile is adaptive—not rigid

Preparing the Final Presentation

Presentation Structure (10–15 mins):

1. Project Summary: What problem did you solve? **2. Team Roles & Dynamics:** Who did what? How did Agile work for you?

3. **Planning Process:** Tools used, how backlog was created
4. **Sprint Demonstrations:** Snapshots or walkthroughs of deliverables
5. **Retrospective Learnings:** What went well? What changed?
6. **Next Steps:** If you had more time, what would you improve?

Format: PowerPoint, live demo, Miro board, or recorded walkthrough

Peer Review and Feedback

Each team member rates peers (anonymous):

- Contribution to sprint tasks
- Collaboration and communication
- Accountability and team spirit

Scoring Format (1–5):

- 1 = Rarely contributed
- 5 = Consistently exceeded expectations

Peer scores can influence **final grades or feedback coaching.**

alok.giri@ncit.edu.np

9.1 Effective Communication – Facilitating Open and Honest Dialogue

Why Communication Matters in Agile

- Agile ceremonies (standups, planning, reviews, retrospectives) rely

heavily on clear communication

- Promotes transparency and trust
- Enables fast feedback and continuous improvement

Characteristics of Agile Communication

- **Open:** Encourages honesty without fear of blame
- **Continuous:** Happens regularly, not just during meetings
- **Respectful:** Differences of opinion are welcomed
- **Inclusive:** Everyone's voice matters

Techniques for Effective Communication

- Practice **active listening**
- Ask **open-ended and clarifying questions**
- Give and receive **constructive feedback**
- Use **non-verbal cues** effectively (especially in virtual settings)

- Maintain **transparency** in status updates and blockers

Common Communication Barriers

- Lack of attention or interest
- Misinterpretation or assumptions
- Language or cultural differences
- Poor listening skills

Understanding Conflict

- Conflict is a natural and sometimes **necessary** part of teamwork
- Can be either **constructive** (positive) or **destructive** (negative)

Positive vs Negative Conflict

- **Positive Conflict:** Leads to innovation, better decisions •

Negative Conflict: Causes stress, poor morale, project delays

Common Sources of Conflict

- Role ambiguity
- Conflicting priorities
- Personality differences
- Communication gaps

Types of Conflict

- **Task conflict** – Disagreement on what/how to do something
- **Relationship conflict** – Personal clashes
- **Process conflict** – Disagreements about how things are done

Strategies to Resolve Conflict

- Address issues **early** before they escalate
- Focus on the **issue**, not the person
- Use **"I" statements** instead of blaming
- Establish **shared goals**
- Use **neutral mediators** if necessary (e.g., Scrum Master)

9.3 Leadership in Agile Teams – Servant Leadership Principles

What is Servant Leadership?

- A leadership style where the leader's role is to **serve the team**
- Focuses on team **empowerment and growth over personal** ●

Listens first, acts second

Principles of Servant Leadership

- **Listening** – Actively understanding team needs
- **Empathy** – Relating to team member challenges
- **Stewardship** – Holding team goals above personal goals
- **Commitment to Growth** – Helping each team member thrive
- **Awareness** – Being mindful of team health and morale
- **Healing**- Help team members recover from conflict or failure
- **Persuasion**- Lead by influence, not authority
- **Building Community**- Foster a sense of belonging and trust

Servant vs Traditional Leadership

Traditional Leadership Servant Leadership

Commanding Style Supportive Style Focus on

Hierarchy Focus on Service Decision-maker

Facilitation

Controls outcomes Enables ownership

Role of Scrum Master as Servant Leader

- Facilitates rather than commands
- Removes team impediments
- Protects team from external pressure
- Encourages collaboration and ownership\

9.4 Building a Collaborative Culture – Fostering Trust and Accountability

Why Collaboration is Crucial

- Agile thrives on **teamwork**, not individual heroics ●
- Promotes **shared understanding and faster delivery** ●
- Encourages **continuous learning and adaptation**

Building Trust Within the Team

- Be **reliable** – keep promises
- Be **honest** – share both good and bad news
- Be **vulnerable** – admit mistakes
- Be **supportive** – give praise and help

Promoting Accountability

- Encourage **shared ownership** of goals and outcomes
- Foster a **blameless culture** – failures are learning opportunities
- Use retrospectives to identify areas of improvement
- Use clear roles and responsibilities

Characteristics of Collaborative Teams

- Open communication
- Mutual respect
- Shared decision-making
- Equal contribution
- Collective ownership of outcomes

Ways to Build Trust in Teams

- Be consistent and follow through on commitments
- Be open and transparent with your work
- Show vulnerability and ask for help when needed
- Support and appreciate team members