QA
Learn. To Change.

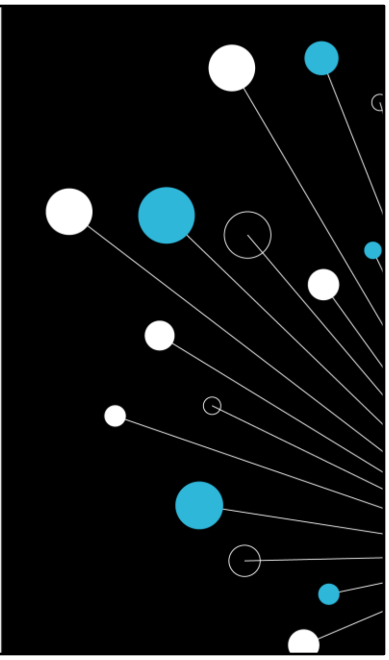
SAMMAN TRAINING

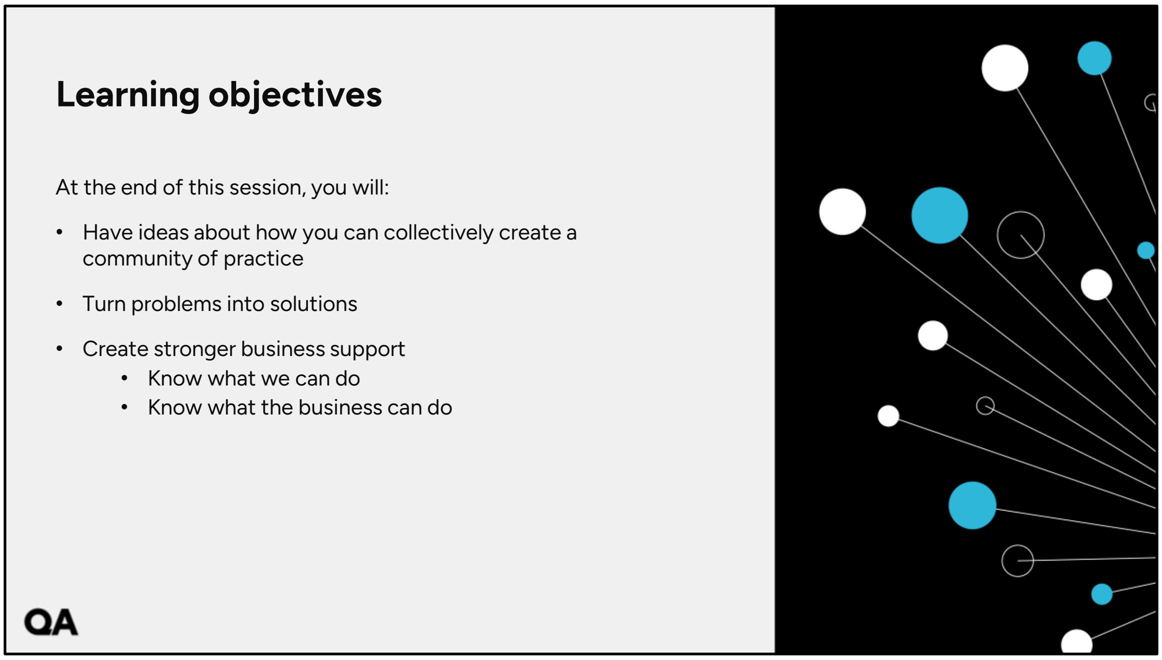
Phase 3 - Creating a Community of Practice

Programme Learning objectives

- Understand the value of the Samman training hour and how to apply it
- Understand how to apply the 4C training model – Connect, Concepts, Concrete Practice and Conclusions
- Understand how mentoring / coaching can help engineers to learn coding skills
- How to set individual and team technical goals
- Develop facilitation skills to run technical workshops on TDD, Refactoring in ensemble and paired set ups
- Be able to contextualise technical content to the team needs
- Learn how to embed continuous learning and experimentation in engineering work

QA



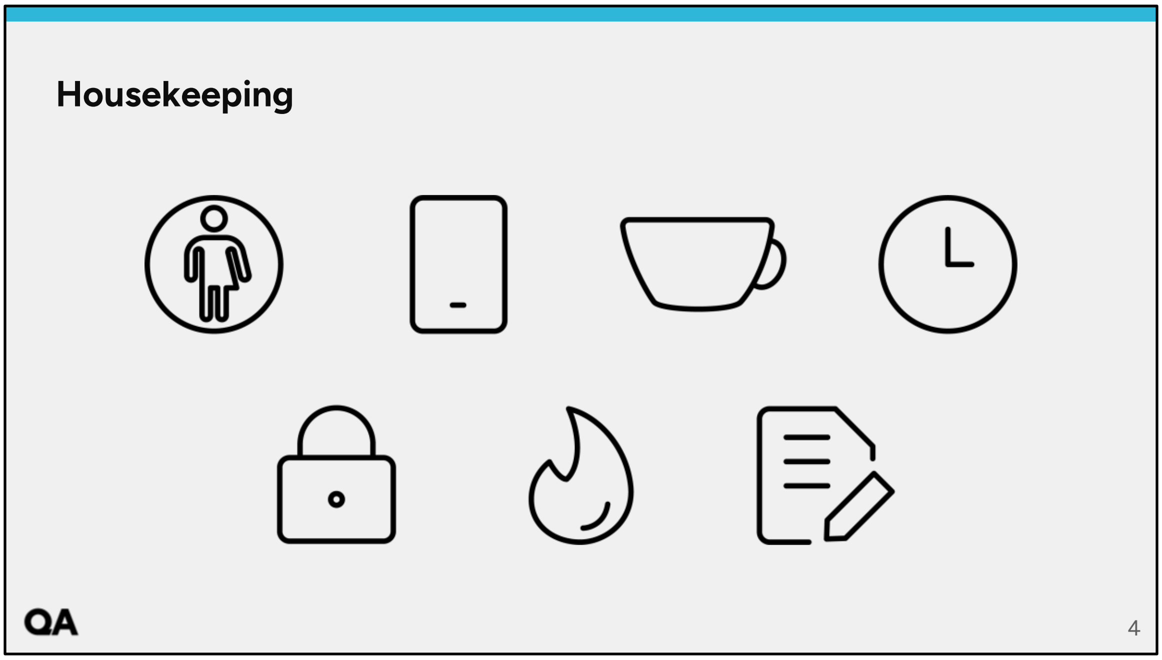


Learning objectives

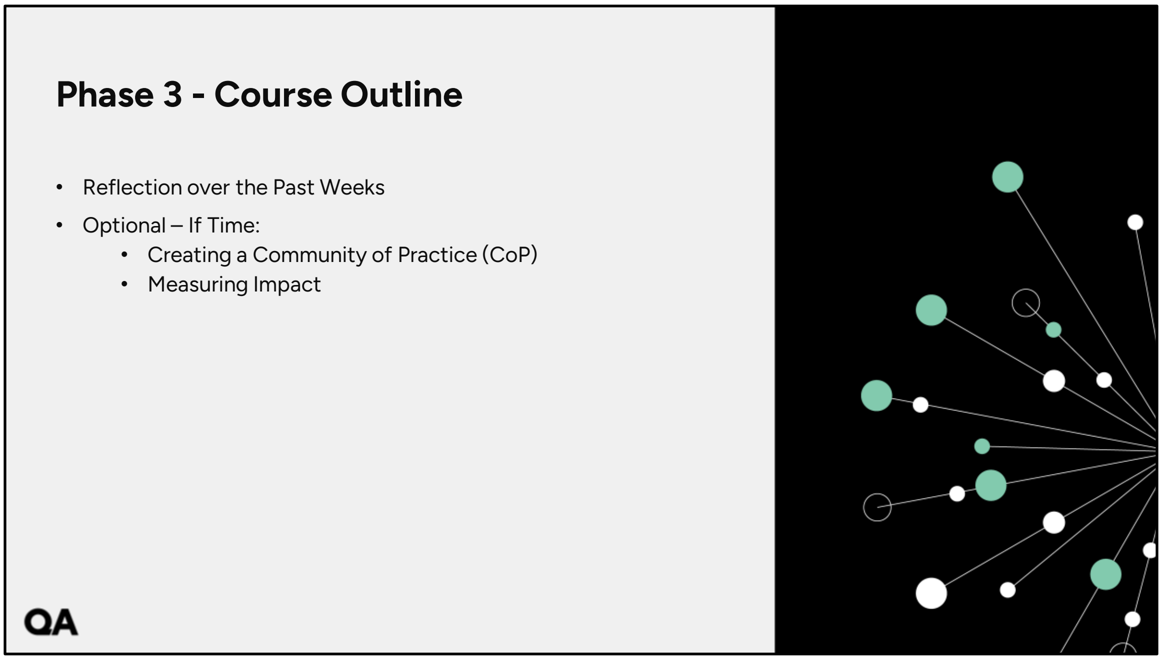
At the end of this session, you will:

- Have ideas about how you can collectively create a community of practice
- Turn problems into solutions
- Create stronger business support
    - Know what we can do
    - Know what the business can do

QA



Housekeeping



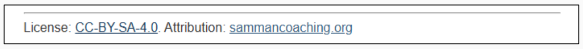
Phase 3 - Course Outline

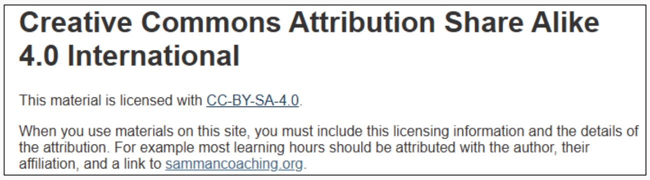
- Reflection over the Past Weeks
- Optional – If Time:
    - Creating a Community of Practice (CoP)
    - Measuring Impact

QA

QA

License and Disclaimer


License: CC-BY-SA-4.0. Attribution: sammancoaching.org

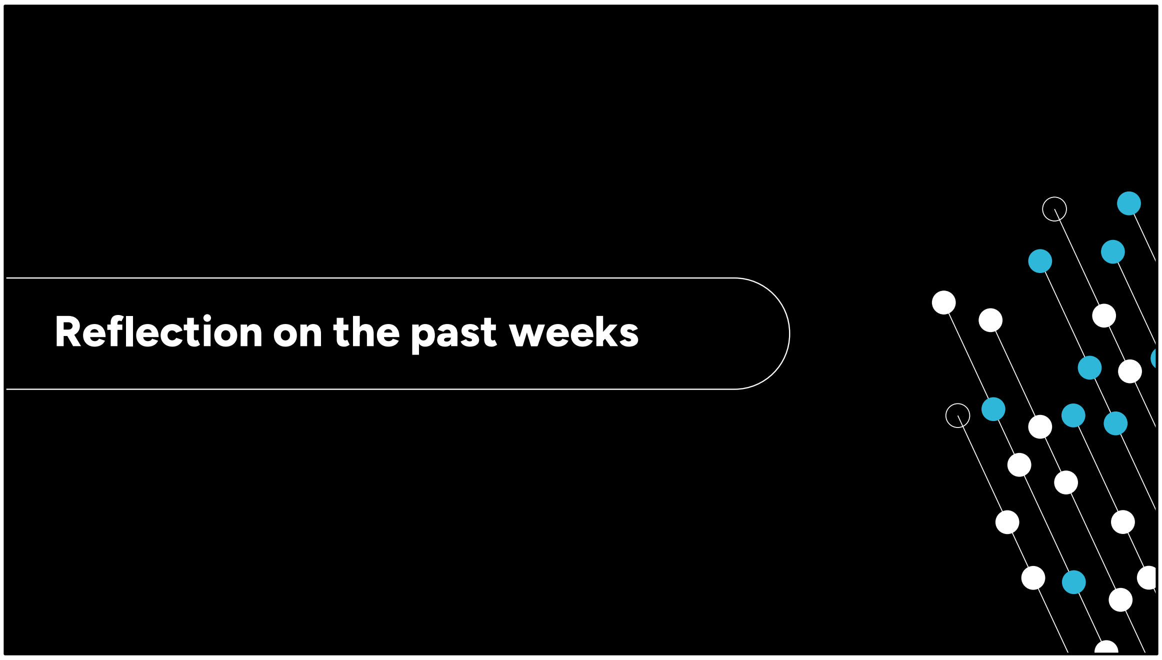

Creative Commons Attribution Share Alike 4.0 International

This material is licensed with CC-BY-SA-4.0.

When you use materials on this site, you must include this licensing information and the details of the attribution. For example most learning hours should be attributed with the author, their affiliation, and a link to sammancoaching.org.

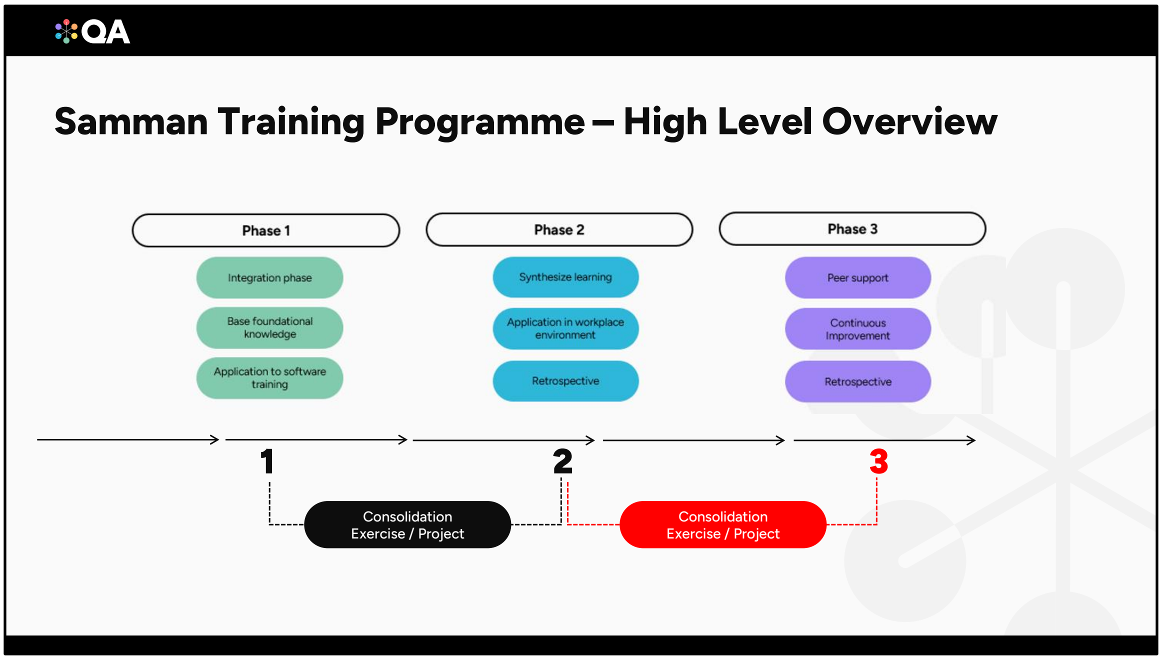
“Samman Coaching” is a trademark owned by the Samman Technical Coaching Society, so you may not use these words in any way that makes it sound like you represent the society or that your training is officially sanctioned by us.

Sammancoaching.or
g



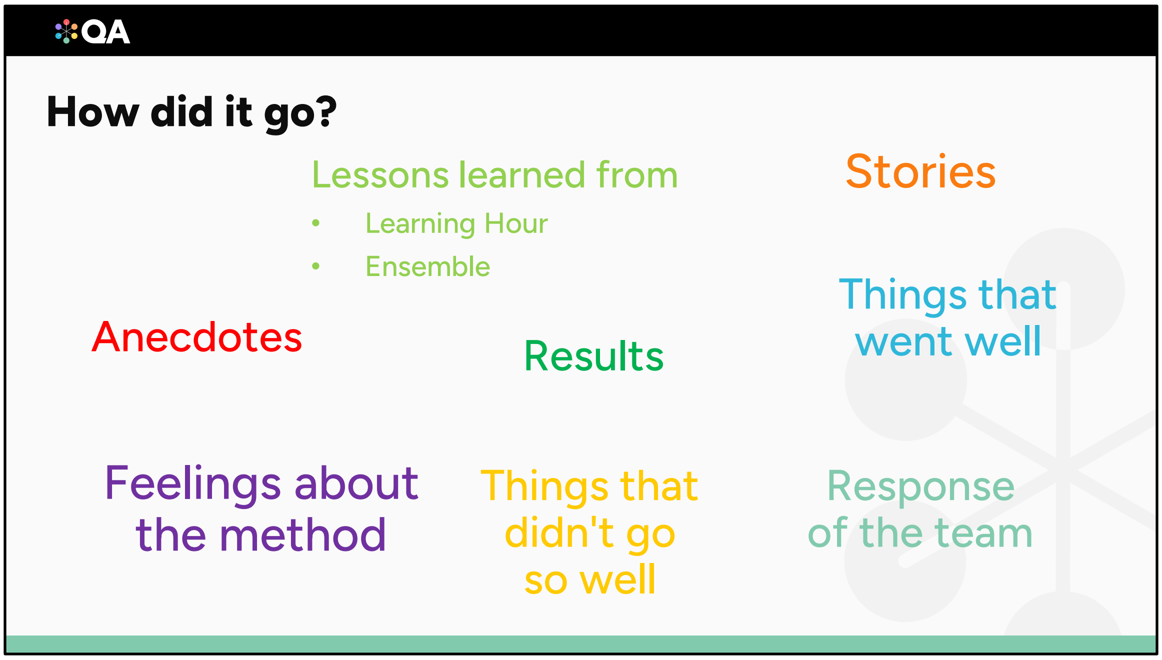
Reflection on the past weeks

Create Break out groups that include people who did and did not manage to run a learning Hour and/or Ensemble.
Get them to discuss the questions on the next 2 slides for 20 mins and then report back.



Samman Training Programme – High Level Overview

Phase 1
Integration phase
Base foundational knowledge
Application to software training
Phase 2
Synthesize learning
Application in workplace environment
Retrospective
Phase 3
Peer support
Continuous Improvement
Retrospective
1
2
3
Consolidation Exercise / Project
Consolidation Exercise / Project



QA

How did it go?

Lessons learned from

- Learning Hour
- Ensemble

Stories

Things that went well

Anecdotes

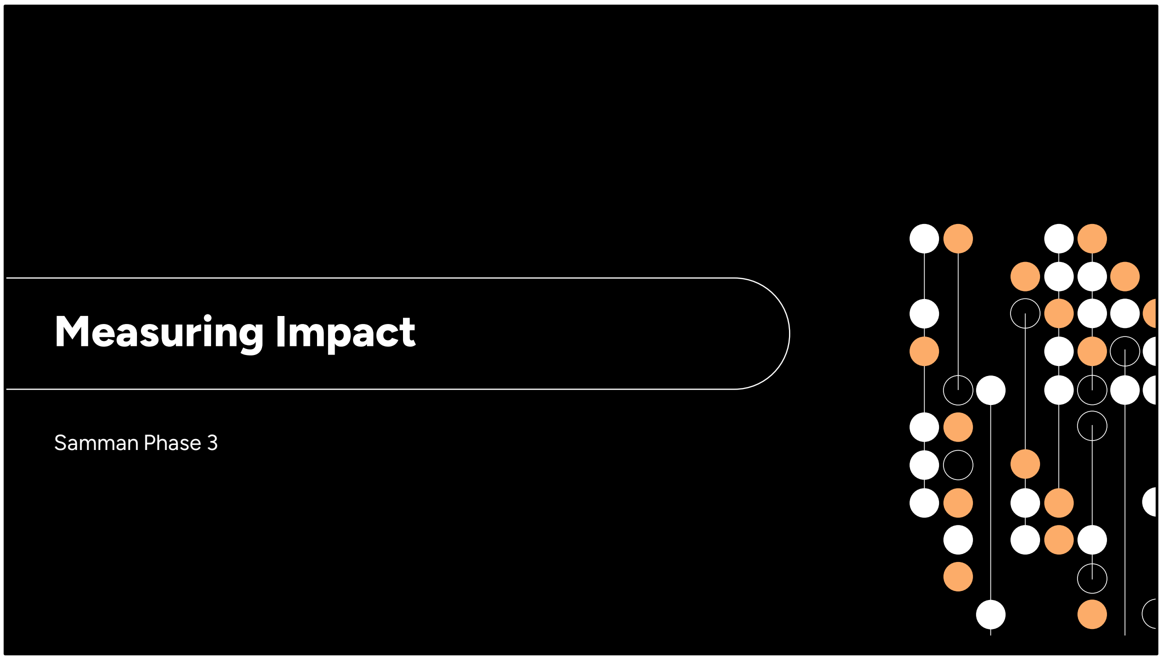
Results

Feelings about the method

Things that didn't go so well

Response of the team

Refer to notes you (may) have made in the "Preparation for Phase 2" document given out at the end of phase 1.

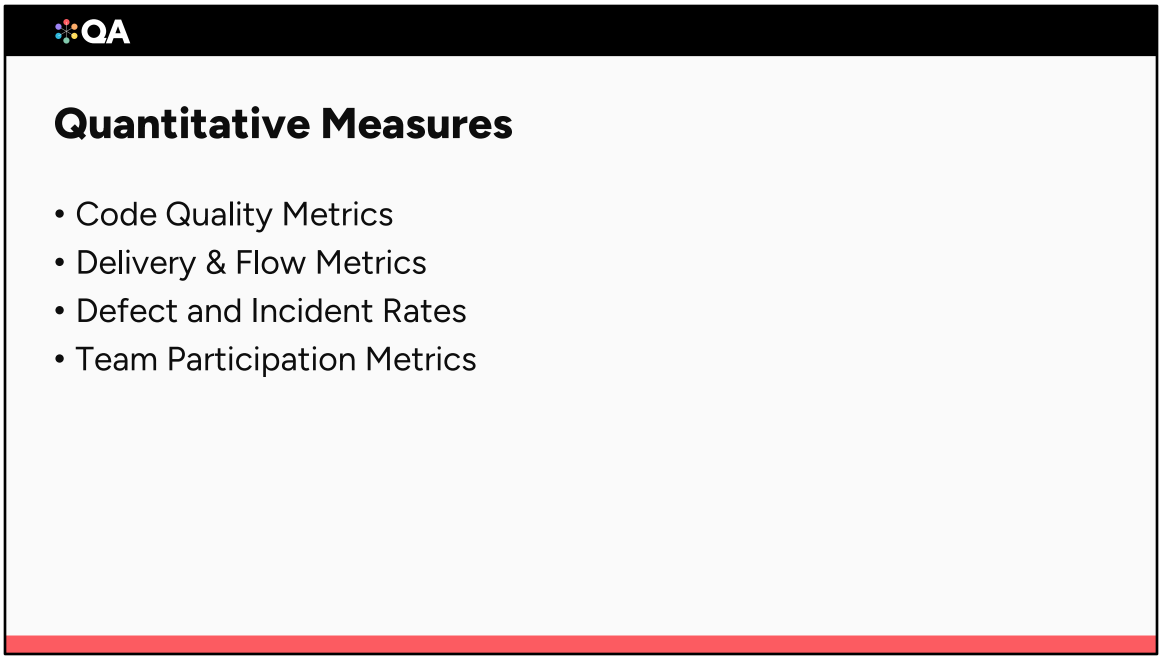


Measuring Impact

Samman Phase 3

Quantitative vs Qualitative Measurements

- Quantitative research focuses on numerical data and statistical analysis to test hypotheses and establish relationships between variables
- Qualitative research explores concepts, experiences, and perspectives through non-numerical data like interviews and observations, aiming for in-depth understanding
- Essentially, quantitative data tells you what happened, while qualitative data helps explain why it happened.



Quantitative Measures

- Code Quality Metrics
- Delivery & Flow Metrics
- Defect and Incident Rates
- Team Participation Metrics

QA

Code Quality Metrics

- **Test Coverage**: Track the percentage of code covered by automated tests. Higher coverage can indicate better practices. Often improves with TDD focus in ensemble programming sessions.
- **Static Analysis Violations** (e.g. SonarQube, ESLint): Reduction over time may suggest better coding habits and shared understanding.
- **Number of Code Smells / Cyclomatic Complexity**: Lower complexity and better structure through collective refactoring.
- **Code Review Metrics**: Monitor time to review and merge pull requests, and the number of review comments.

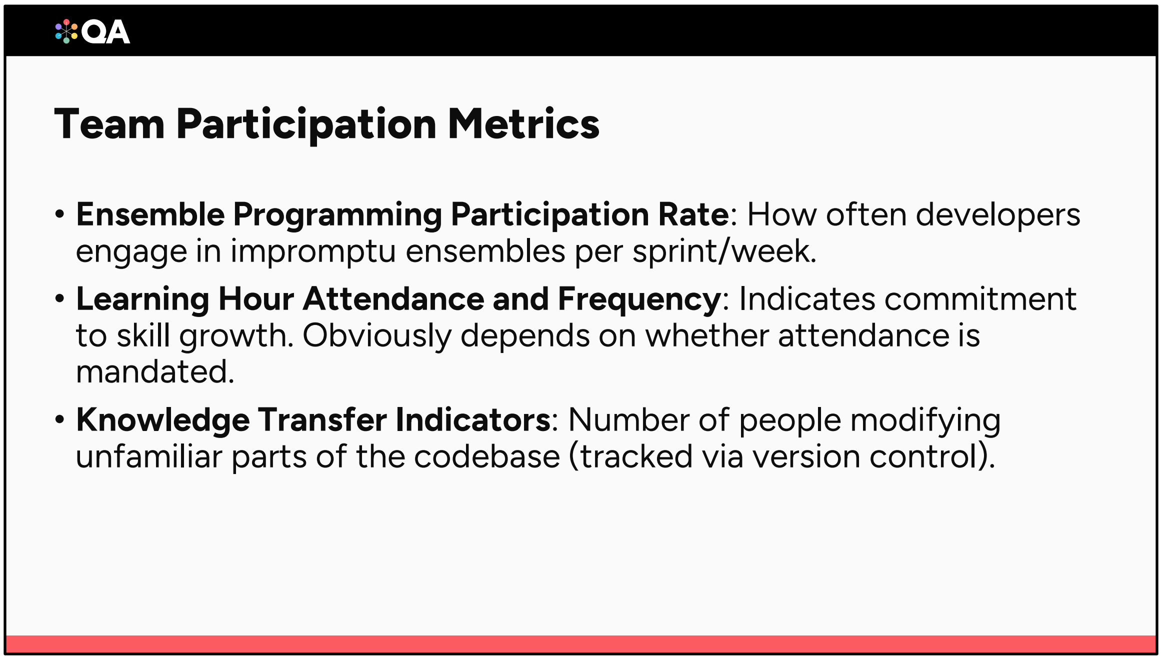
QA

Delivery & Flow Metrics

- **Lead Time**: Track the time from feature request to production release. Teams may deliver faster as shared ownership grows.
- **Change Failure Rate**: Production incidents or rollbacks. Improved collaboration and testing should reduce this.
- **Deployment Frequency**: Teams may gain confidence to deploy more often with shared learning and cleaner code. Increased frequency often indicates smoother workflows.
- **Pull Request (PR) Size and Review Cycle Time**: Shorter, more frequent PRs or fewer PRs if more code is written collaboratively. Measure the time taken to complete user stories or tasks.
- **Team Velocity**: Use story points or completed tasks per sprint to gauge productivity.

Defect and Incident Rates

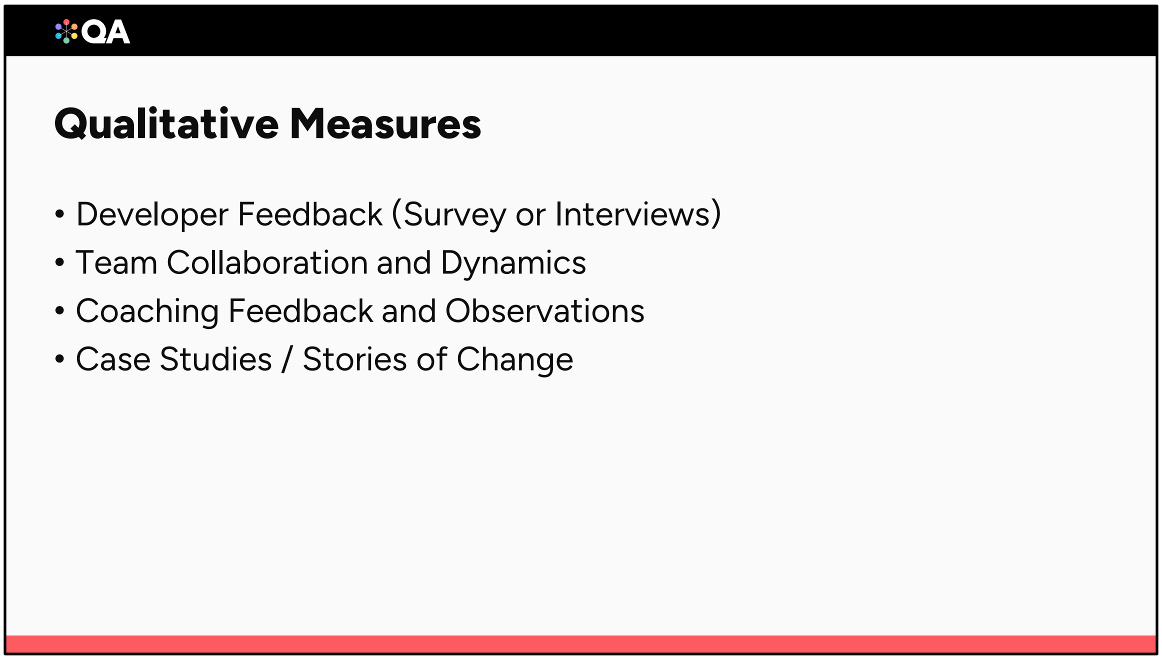
- **Bug Rate Post-Release**: Often decreases as code is more thoroughly discussed and tested in ensembles. Fewer defects suggest better code quality.
- **Escaped Defects**: Monitor production bug counts per sprint or per release.



QA

Team Participation Metrics

- **Ensemble Programming Participation Rate**: How often developers engage in impromptu ensembles per sprint/week.
- **Learning Hour Attendance and Frequency**: Indicates commitment to skill growth. Obviously depends on whether attendance is mandated.
- **Knowledge Transfer Indicators**: Number of people modifying unfamiliar parts of the codebase (tracked via version control).



QA

Qualitative Measures

- Developer Feedback (Survey or Interviews)
- Team Collaboration and Dynamics
- Coaching Feedback and Observations
- Case Studies / Stories of Change

Developer Satisfaction (Survey or Interviews)

Ask before and after adopting Samman techniques:

- How confident are you in the team's technical skills?
- How safe do you feel raising code quality concerns?
- How much do you learn from others in your team?
- How effective is our team at improving the design of our code over time?
- **Code Quality Perception**: Gather feedback on maintainability, readability, and technical debt from developers.

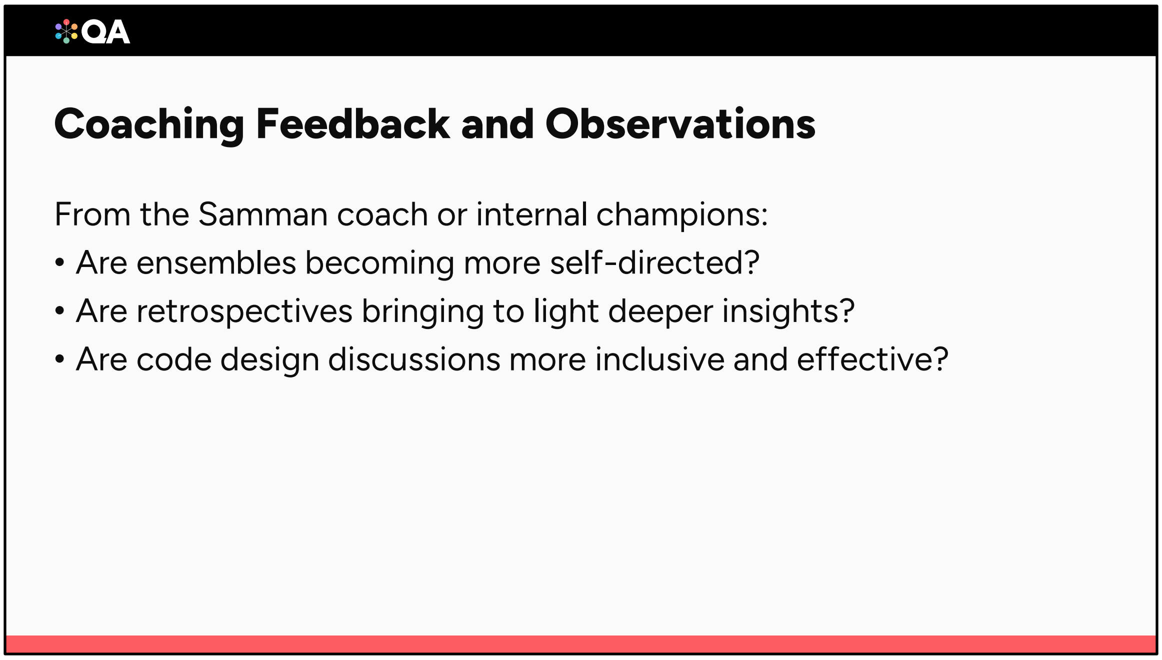
Consider repeating this every 3–6 months.

QA

Team Collaboration and Dynamics

- **Psychological Safety**: Are people more willing to pair, ask questions, or admit mistakes?
- **Cross-functional Engagement**: Are developers collaborating more with Quality Assurance, User Experience and/or Product Management?
- **Onboarding Time**: Are new developers ramping up faster due to collective code ownership?

Gather the information via retrospectives and feedback sessions.



Coaching Feedback and Observations

From the Samman coach or internal champions:

- Are ensembles becoming more self-directed?
- Are retrospectives bringing to light deeper insights?
- Are code design discussions more inclusive and effective?

In the context of Samman Technical Coaching, ensemble programming starts with facilitated sessions led by a coach. Over time, the goal is for the team to internalize the practice and own the process themselves. When ensembles become "self-directed," it means:

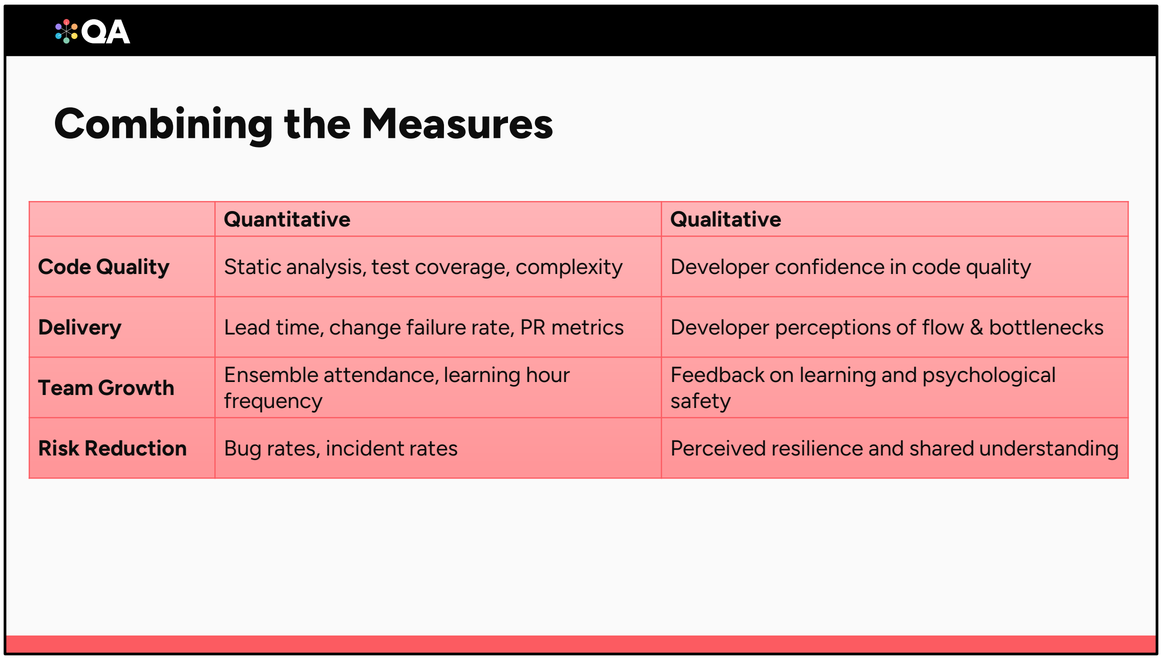
- The team can organize and run ensemble programming sessions without relying on a coach.
- They rotate roles (driver, designated navigator, navigator) effectively on their own.
- They manage their collaboration dynamics, timeboxing, and discussions without prompting.
- They set goals for the session, track progress, and retrospect on outcomes—without external facilitation.
- They take initiative to use ensembles for tasks like refactoring, onboarding, or solving tough bugs.

Case Studies / Stories of Change

Document anecdotal wins:

- “We refactored a legacy module safely using TDD in ensemble sessions.”
- “Our last sprint had no production bugs—first time in months.”
- “A junior developer led the ensemble and taught a new testing strategy.”

Collect input from product owners, QA, and other teams on perceived improvements.



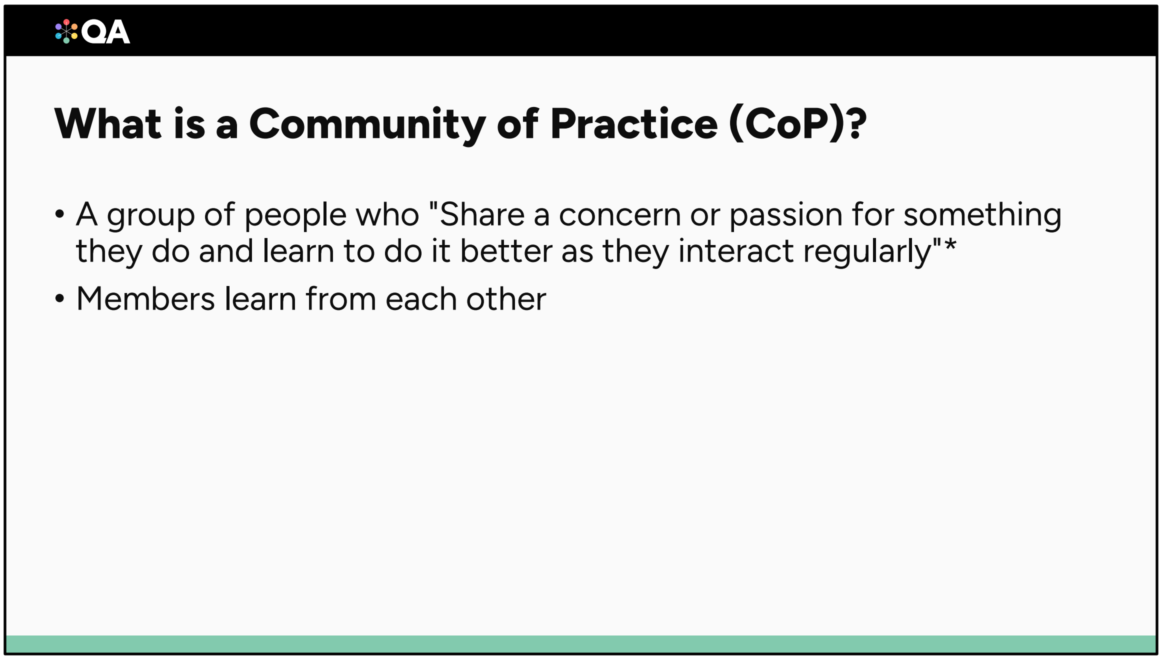
QA

Combining the Measures

	Quantitative	Qualitative
Code Quality	Static analysis, test coverage, complexity	Developer confidence in code quality
Delivery	Lead time, change failure rate, PR metrics	Developer perceptions of flow & bottlenecks
Team Growth	Ensemble attendance, learning hour frequency	Feedback on learning and psychological safety
Risk Reduction	Bug rates, incident rates	Perceived resilience and shared understanding

Creating a Community of Practice (CoP)

Samman Phase 3



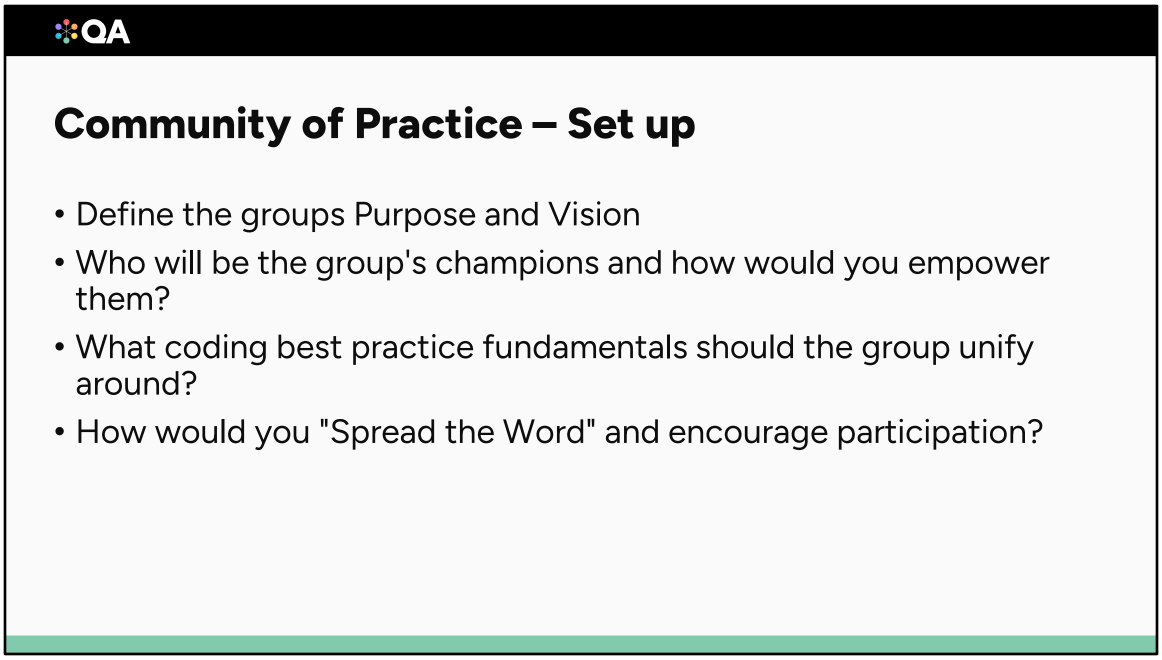
What is a Community of Practice (CoP)?

- A group of people who "Share a concern or passion for something they do and learn to do it better as they interact regularly"*
- Members learn from each other

* Community of practice - Wikipedia

Structural Characteristics

- Domain of knowledge
  - Creates:
    - common ground
    - inspires participation
    - guides learning
    - gives meaning to the actions of the individuals and community.
- Notion of Community
  - creates the social fabric for learning by fostering interactions, encouraging collaboration and the sharing of ideas
- Practice
  - the focus around which the community develops, shares and maintains its core of knowledge



Community of Practice – Set up

- Define the groups Purpose and Vision
- Who will be the group's champions and how would you empower them?
- What coding best practice fundamentals should the group unify around?
- How would you "Spread the Word" and encourage participation?

[* Community of practice - Wikipedia]

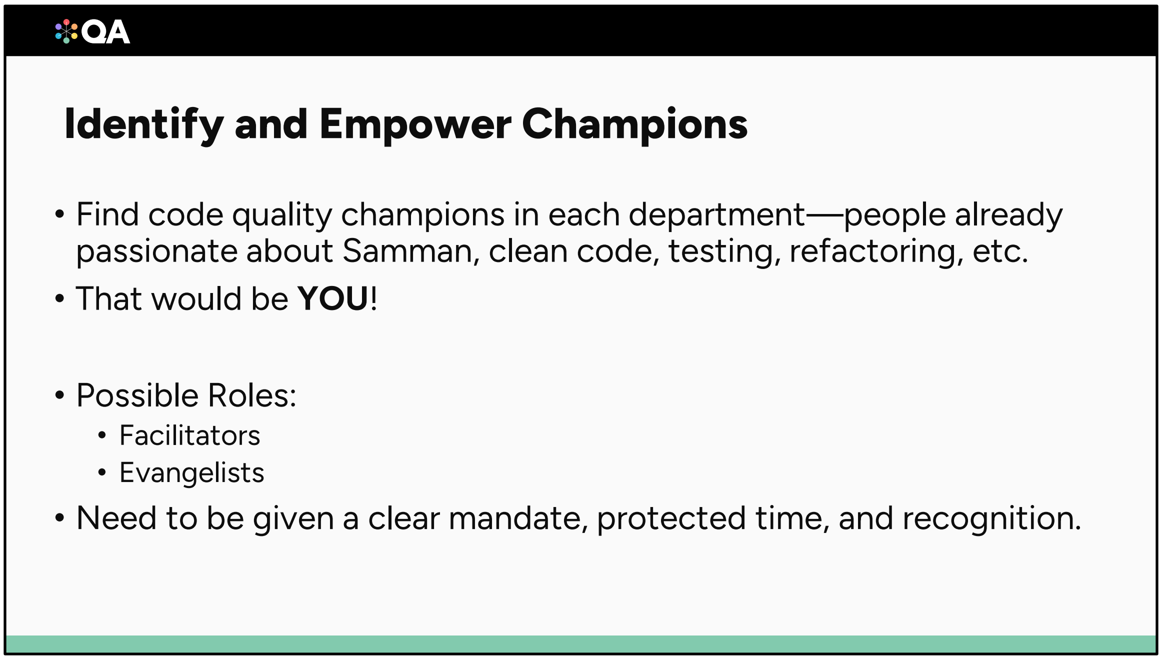
Define the Purpose and Vision

How about:

"To foster a culture of continuous technical learning and collaboration across the organisation by sharing knowledge, practices, tools and metrics via the adoption of Samman Technical Coaching methods."

Key considerations:

- Align with the org's engineering goals (e.g. reliability, maintainability, velocity).
- Frame the CoP as a cross-cutting enabler, not a top-down directive.



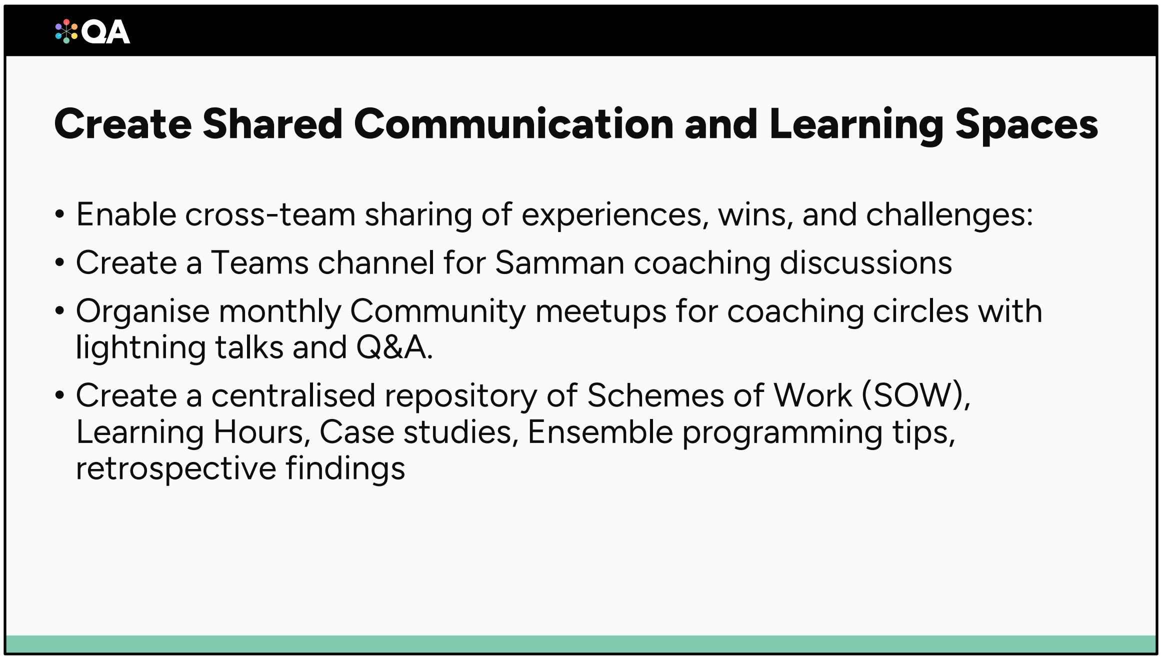
Identify and Empower Champions

- Find code quality champions in each department—people already passionate about Samman, clean code, testing, refactoring, etc.
- That would be **YOU**!

- Possible Roles:
  - Facilitators
  - Evangelists
- Need to be given a clear mandate, protected time, and recognition.

Encourage a Practice Framework

- Encourage weekly or bi-weekly **ensemble programming sessions**.
- Block time for regular **learning hours** (e.g. Test-Driven Development, refactoring, clean code).
- Create new/adapt code Katas that represent the programming languages used by the organisation
- Share lightweight templates:
  - Ensemble programming rotation protocols.
  - Learning hour formats.
  - Reflection and feedback guidelines.



QA

Create Shared Communication and Learning Spaces

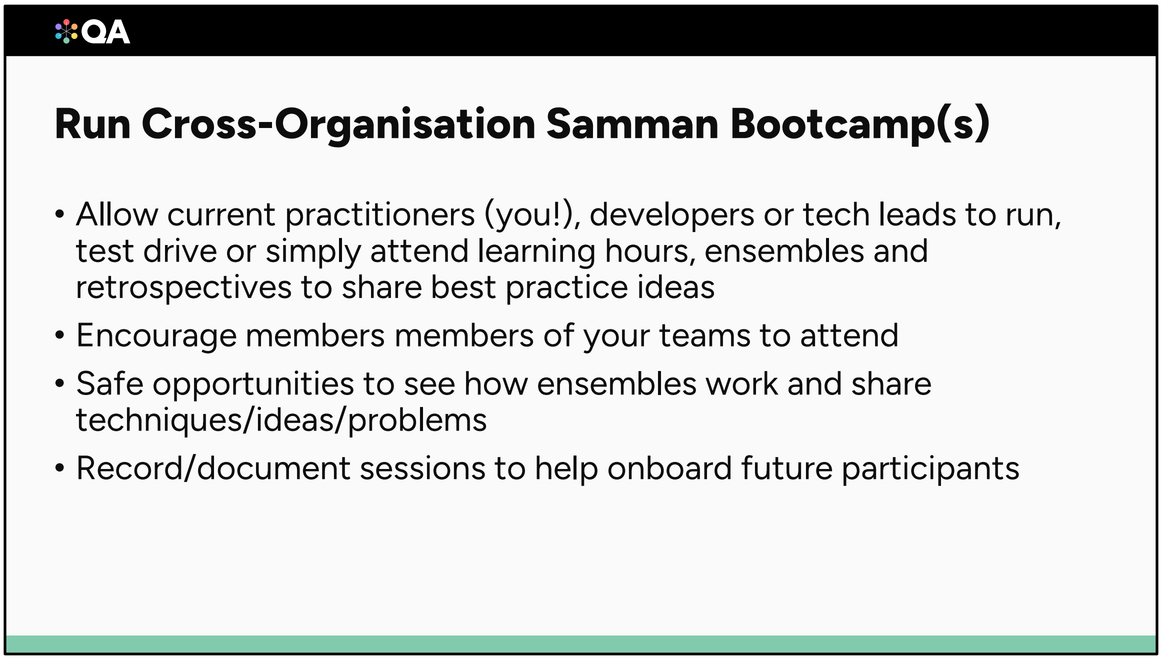
- Enable cross-team sharing of experiences, wins, and challenges:
- Create a Teams channel for Samman coaching discussions
- Organise monthly Community meetups for coaching circles with lightning talks and Q&A.
- Create a centralised repository of Schemes of Work (SOW), Learning Hours, Case studies, Ensemble programming tips, retrospective findings

QA

Create Peer Coaching Groups

Encourage coaches and ambassadors to support each other:

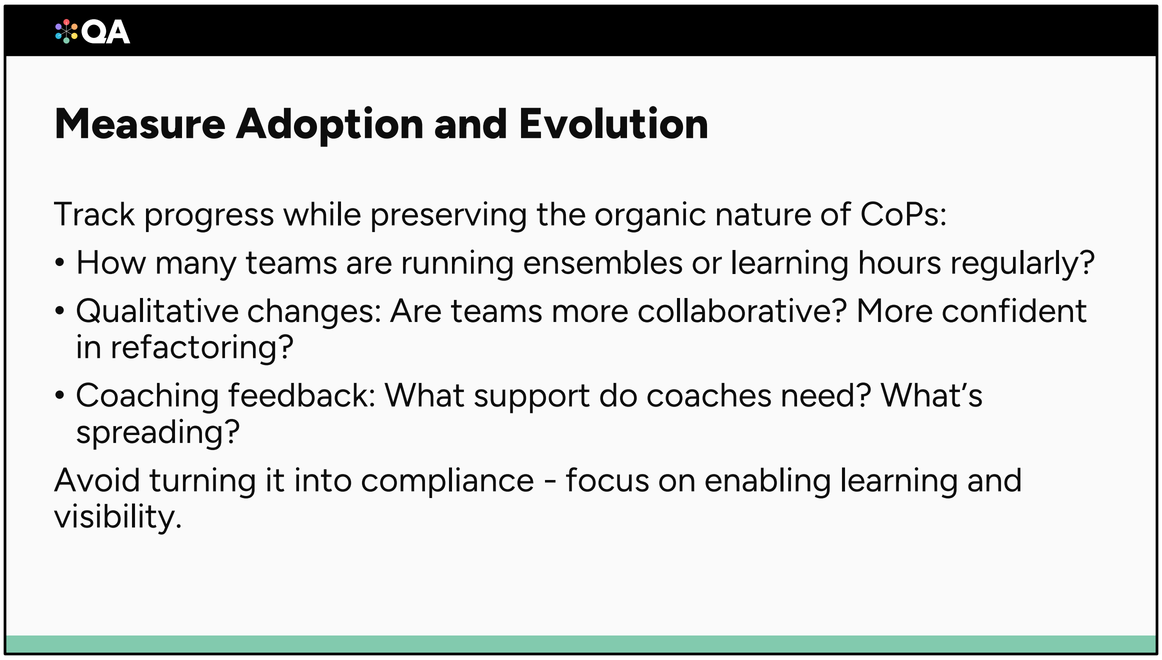
- Regular **peer reflection sessions** (e.g. “coaching dojos”).
- Share what worked and what didn’t.
- Practice facilitation, rotation management, and conflict resolution skills.



QA

Run Cross-Organisation Samman Bootcamp(s)

- Allow current practitioners (you!), developers or tech leads to run, test drive or simply attend learning hours, ensembles and retrospectives to share best practice ideas
- Encourage members members of your teams to attend
- Safe opportunities to see how ensembles work and share techniques/ideas/problems
- Record/document sessions to help onboard future participants


QA

Measure Adoption and Evolution

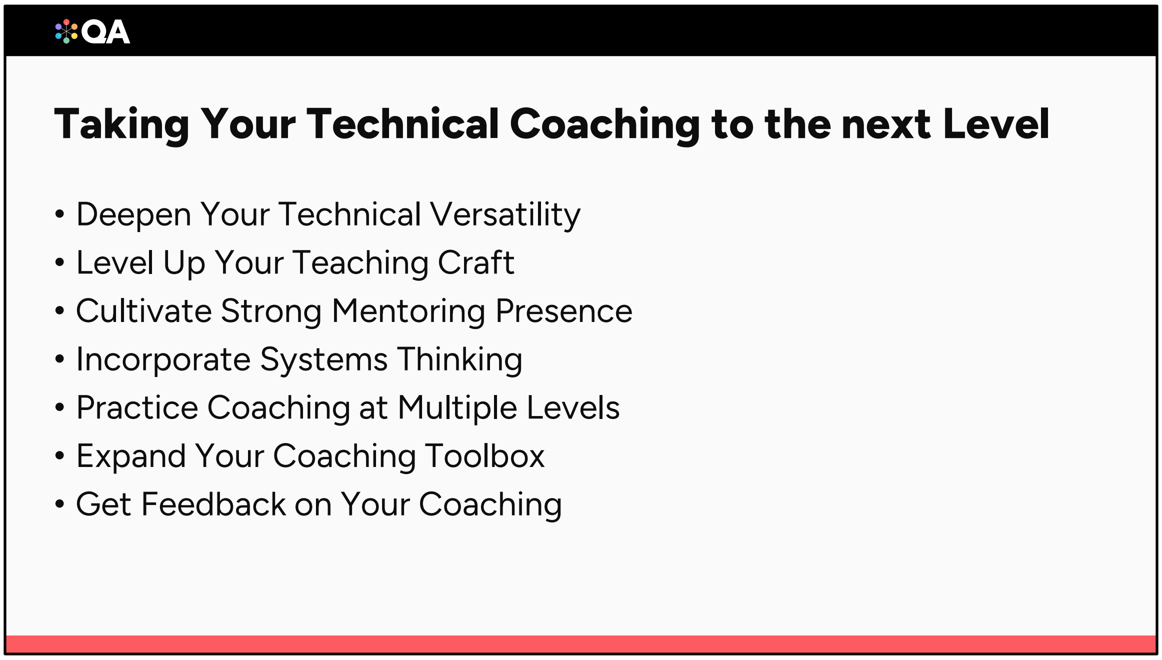
Track progress while preserving the organic nature of CoPs:

- How many teams are running ensembles or learning hours regularly?
- Qualitative changes: Are teams more collaborative? More confident in refactoring?
- Coaching feedback: What support do coaches need? What’s spreading?

Avoid turning it into compliance - focus on enabling learning and visibility.

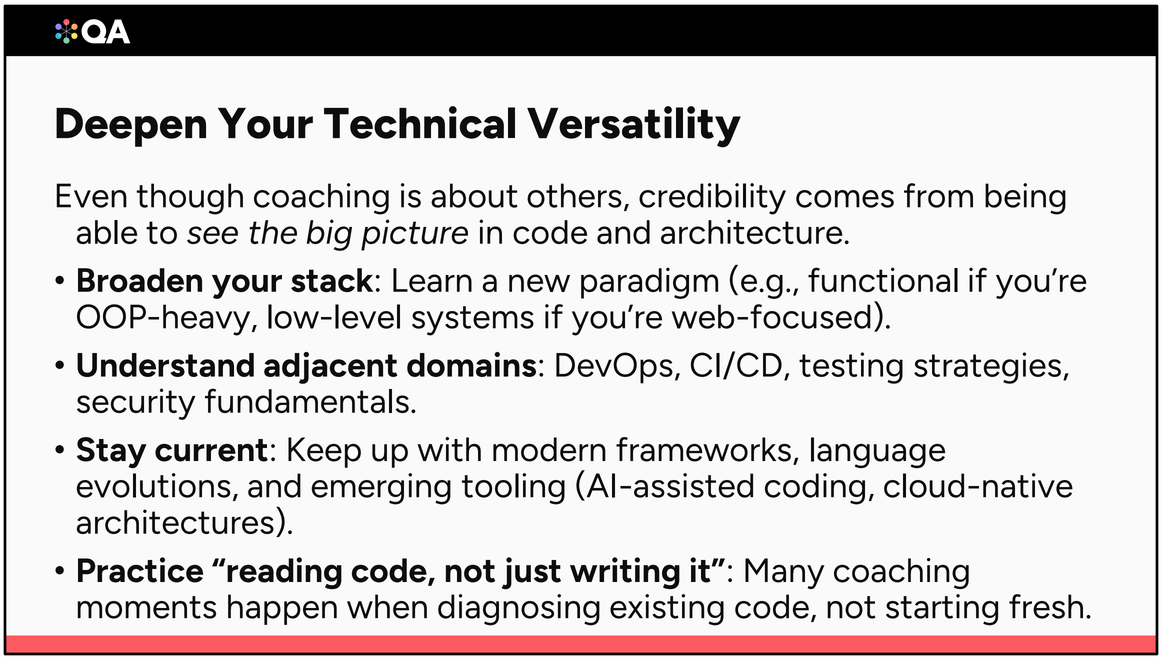
Broadening Coaching Skills

Samman Phase 3



Taking Your Technical Coaching to the next Level

- Deepen Your Technical Versatility
- Level Up Your Teaching Craft
- Cultivate Strong Mentoring Presence
- Incorporate Systems Thinking
- Practice Coaching at Multiple Levels
- Expand Your Coaching Toolbox
- Get Feedback on Your Coaching

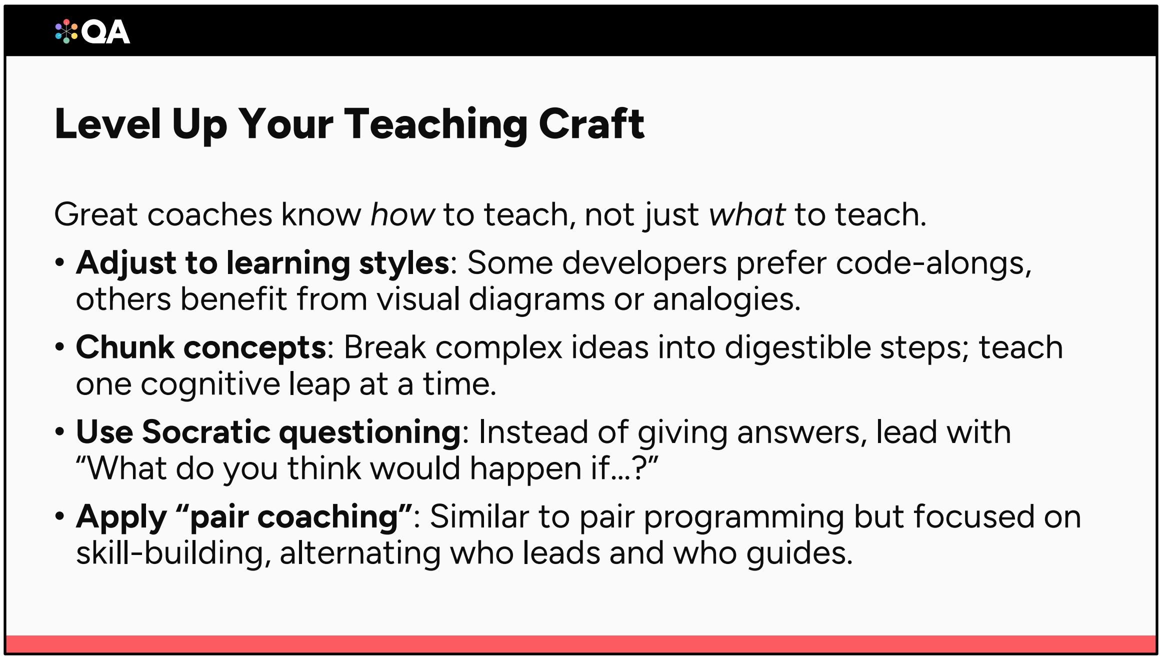


QA

Deepen Your Technical Versatility

Even though coaching is about others, credibility comes from being able to *see the big picture* in code and architecture.

- **Broaden your stack**: Learn a new paradigm (e.g., functional if you're OOP-heavy, low-level systems if you're web-focused).
- **Understand adjacent domains**: DevOps, CI/CD, testing strategies, security fundamentals.
- **Stay current**: Keep up with modern frameworks, language evolutions, and emerging tooling (AI-assisted coding, cloud-native architectures).
- **Practice "reading code, not just writing it"**: Many coaching moments happen when diagnosing existing code, not starting fresh.


QA

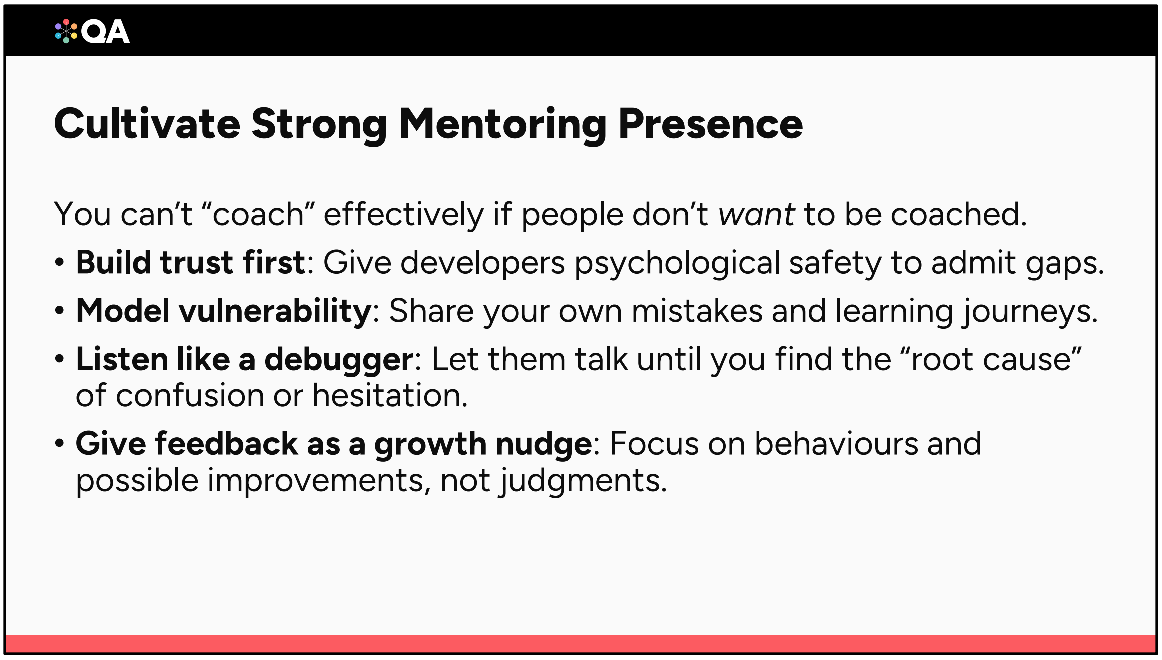
Level Up Your Teaching Craft

Great coaches know *how* to teach, not just *what* to teach.

- **Adjust to learning styles**: Some developers prefer code-alongs, others benefit from visual diagrams or analogies.
- **Chunk concepts**: Break complex ideas into digestible steps; teach one cognitive leap at a time.
- **Use Socratic questioning**: Instead of giving answers, lead with "What do you think would happen if…?"
- **Apply "pair coaching"**: Similar to pair programming but focused on skill-building, alternating who leads and who guides.

Coaching technical practices – Challenges
https://www.infoq.com/articles/coaching-technical-practices/?utm_source=chatgpt.com#:~:text=apply%20it%20correctly.-,Scenarios,-In%20my%20work

Pair Coaching:
The Value of Pair-Coaching... What's in It for You? | Inspect & Adapt
https://www.inspectandadapt.com/blog/pair-coaching#:~:text=Pair%20coaching%20refers

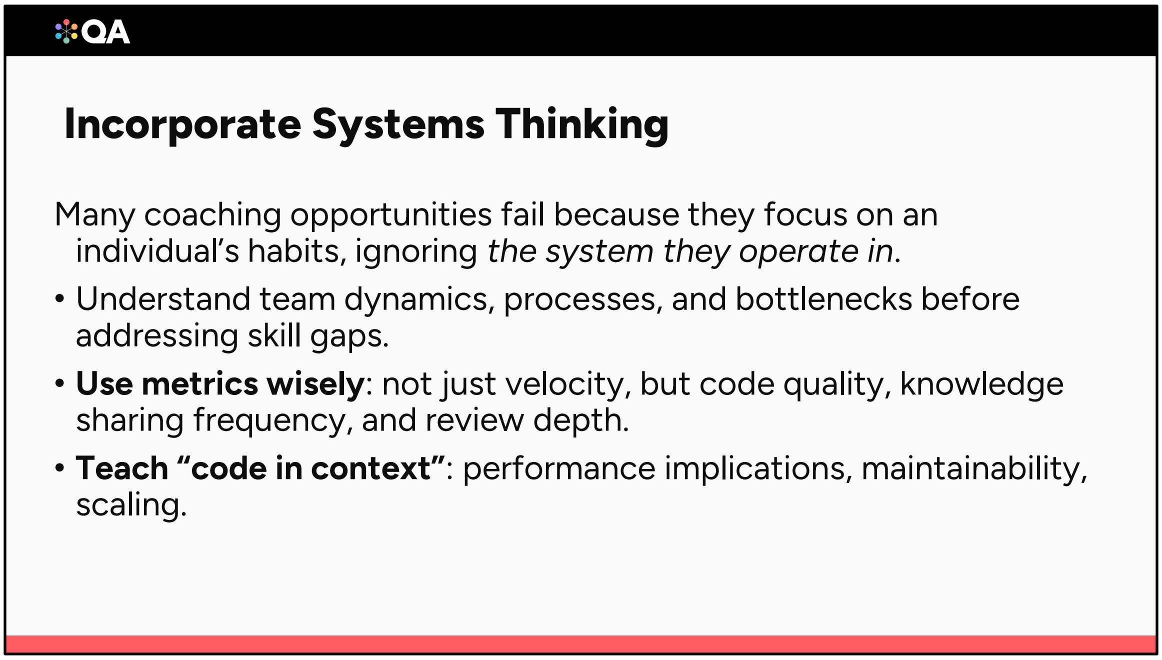


QA

Cultivate Strong Mentoring Presence

You can't "coach" effectively if people don't *want* to be coached.

- **Build trust first**: Give developers psychological safety to admit gaps.
- **Model vulnerability**: Share your own mistakes and learning journeys.
- **Listen like a debugger**: Let them talk until you find the "root cause" of confusion or hesitation.
- **Give feedback as a growth nudge**: Focus on behaviours and possible improvements, not judgments.


QA

Incorporate Systems Thinking

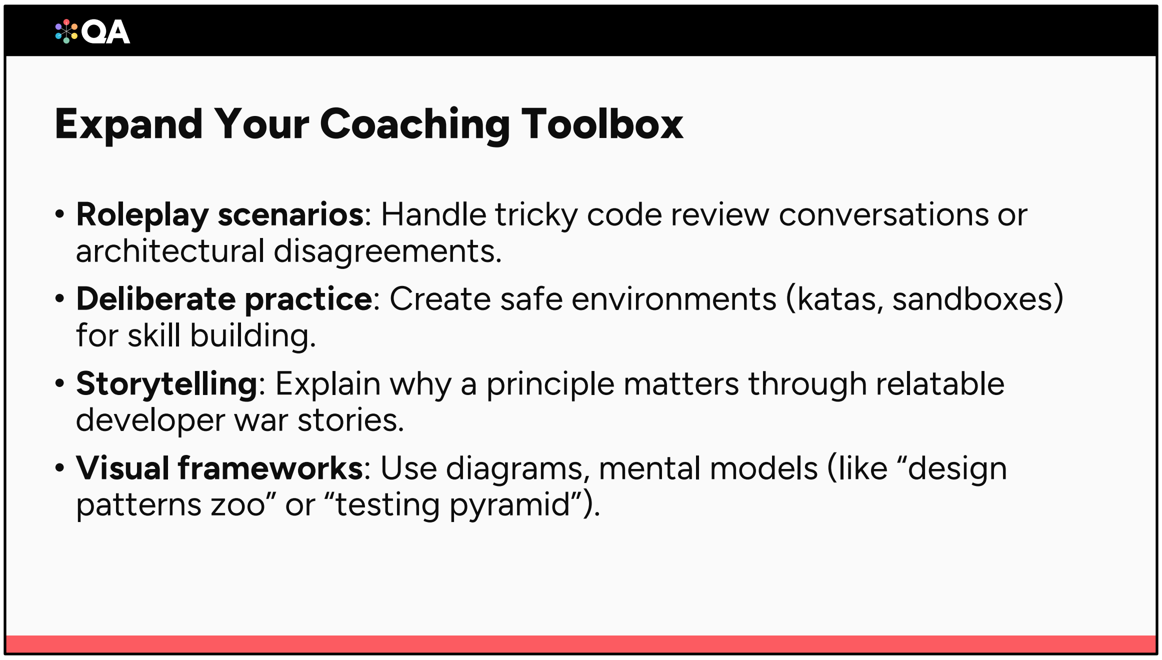
Many coaching opportunities fail because they focus on an individual's habits, ignoring *the system they operate in*.

- Understand team dynamics, processes, and bottlenecks before addressing skill gaps.
- **Use metrics wisely**: not just velocity, but code quality, knowledge sharing frequency, and review depth.
- **Teach "code in context"**: performance implications, maintainability, scaling.

Workplace Coaching And Systems Thinking: Foundations Of Innovation
https://www.forbes.com/councils/forbescoachescouncil/2023/08/25/workplace-coaching-and-systems-thinking-foundations-of-innovation/

Practice Coaching at Multiple Levels

- **Micro level**: One-on-one mentoring, live debugging, code reviews.
- **Team level**: Learning hours, Facilitating ensemble programming, refactoring workshops, test automation initiatives.
- **Organizational level**: Influencing coding standards, architectural principles, and engineering culture


QA

Expand Your Coaching Toolbox

- **Roleplay scenarios**: Handle tricky code review conversations or architectural disagreements.
- **Deliberate practice**: Create safe environments (katas, sandboxes) for skill building.
- **Storytelling**: Explain why a principle matters through relatable developer war stories.
- **Visual frameworks**: Use diagrams, mental models (like “design patterns zoo” or “testing pyramid”).

1. “Don’t Repeat Yourself” (DRY)
War Story:
At a startup, the login logic was copy-pasted across three different services. When the product manager asked for “just one little change” to password rules, only two services got updated. A week later, support tickets piled up: some users could log in to the web app but not the mobile app. Developers spent days untangling the inconsistency.
Lesson: Repetition isn’t just extra code — it’s extra ways for your system to drift apart, silently multiplying bugs.
2. “You Build It, You Run It” (accountability in DevOps)
War Story:
A backend engineer once shipped a feature with a clever but untested caching strategy. At 2 a.m., the system melted down under load. The on-call ops engineer (who hadn’t even touched the code) had no clue why the cache was behaving that way. After a few of these incidents, leadership switched to a “developers own their code in production” model. Suddenly, testing and observability became a priority — because no one wants to debug mysterious outages half-asleep.
Lesson: When you’re the one holding the pager, “good enough” code suddenly isn’t good enough.
3. “Premature Optimization is the Root of All Evil”
War Story:
A team spent two sprints rewriting a data pipeline with exotic concurrency tricks to shave milliseconds. Turns out the real bottleneck wasn’t CPU speed — it was waiting on an external API. Meanwhile, the customer-facing features were slipping, because engineers were busy

"optimizing" the wrong thing.
Lesson: Measure first, optimize later. Otherwise, you're polishing the hubcaps on a car with no engine.

4. "Code is Read More Often than Written"

War Story:
A contractor once shipped a "brilliant" piece of regex magic that parsed edge-case inputs. It worked. But when he left, the next dev inherited a bug ticket, stared at the regex wall for an hour, and gave up, writing a hacky workaround. Over time, the system became a pile of hacks on hacks — because no one could understand the original code.
Lesson: Clever code impresses once; clear code saves time forever.

5. "Fail Fast"

War Story:
An e-commerce checkout service silently swallowed payment errors. Instead of failing loudly, it just logged a warning. Customers thought their orders went through, but the payments were actually rejected. By the time finance caught on, reconciling accounts was a nightmare. After that, the team redesigned the service to fail visibly and immediately if something went wrong.
Lesson: Early pain is better than hidden chaos later.

Design Pattern Zoo

Relatable war story example:
On one project, a junior dev fresh out of a design-patterns-heavy course proudly refactored a simple file uploader into a "zoo":
•An *Abstract Factory* to create the uploader,
•A *Singleton* for configuration,
•A *Strategy* for picking storage backends,
•An *Observer* for progress updates.
The end result? Five times as many files, harder to test, and nobody could tell where to make a small tweak. The team had to strip half of it out just to ship.
Moral: A design patterns zoo is fun to tour, but you don't want to live in one. Patterns are powerful when solving real problems, but painful when they're applied just for show.

Testing Pyramid

The **testing pyramid** is a mental model that helps teams think about how to structure automated tests in a way that's both effective and sustainable.
At its core, the pyramid says:
•**Lots of fast, low-level tests at the bottom**
•**Fewer, slower, high-level tests at the top**
Picture a triangle:

Base: Unit Tests
•Small, fast, cheap to run.
•Test individual functions, classes, or modules in isolation.
•Example: "Does calculateDiscount(10, GOLD) return the right number?"
•You want thousands of these, because they give quick feedback.

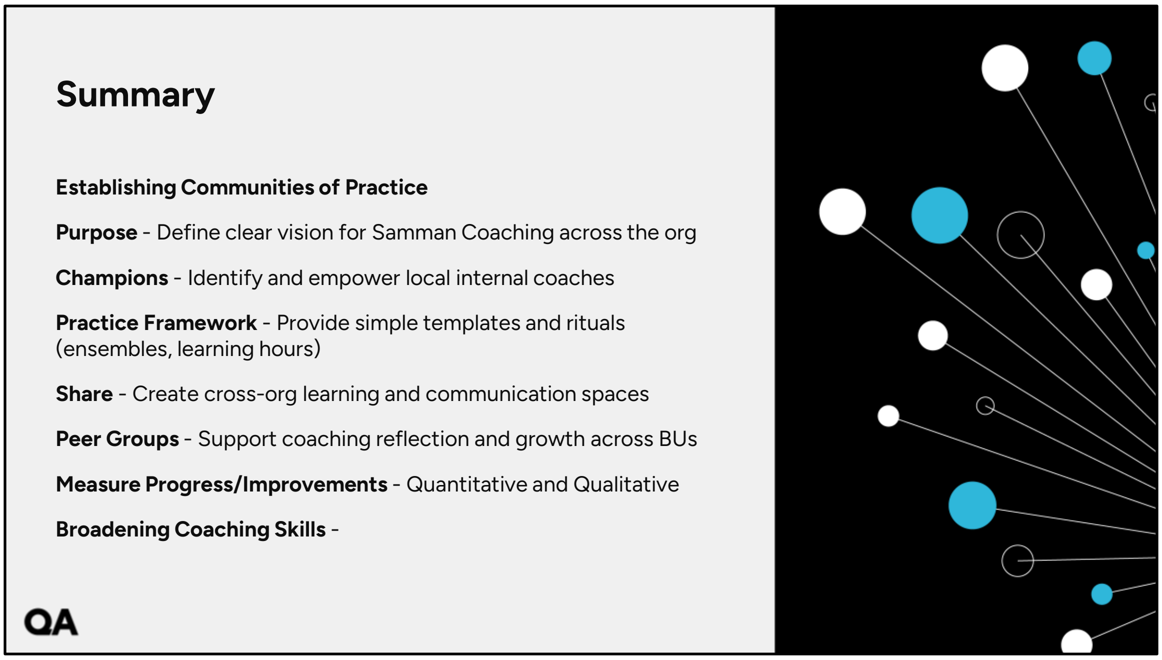
Middle: Integration/Service Tests
•Test how components work together.
•Example: "Does the OrderService correctly call the PaymentService and handle the response?"
•Slower and more fragile than unit tests, so you have fewer of them.

Top: End-to-End (UI/Acceptance) Tests
•Test the whole system from the user's perspective.
•Example: "Can a customer log in, add an item to the cart, and check out successfully?"

•Very slow and brittle, so you keep them minimal — just enough to prove the most critical flows work.
Why it matters (war story style):
A team I worked with once flipped the pyramid upside down:
•They had dozens of Selenium end-to-end tests for every edge case.
•Each test run took **hours**.
•Half the tests were flaky (failing randomly).
•Developers stopped running them, because they slowed down CI.
When a bug slipped through, people blamed QA, not the process.
Later, they rebuilt the test suite following the pyramid model:
•Tons of unit tests for logic.
•A smaller set of API/service tests.
•A lean set of “golden path” end-to-end tests.
Result: Test runs went from hours to minutes, CI actually gave fast feedback, and developers trusted the safety net again.

Get Feedback on Your Coaching

- Record coaching sessions (with consent) and self-review.
- **Ask your coachees**: *"What part of our session was most helpful? Least helpful?"*
- **Find a *coach for the coach***: join a community of tech mentors to exchange tactics.



Summary

Establishing Communities of Practice

Purpose - Define clear vision for Samman Coaching across the org

Champions - Identify and empower local internal coaches

Practice Framework - Provide simple templates and rituals (ensembles, learning hours)

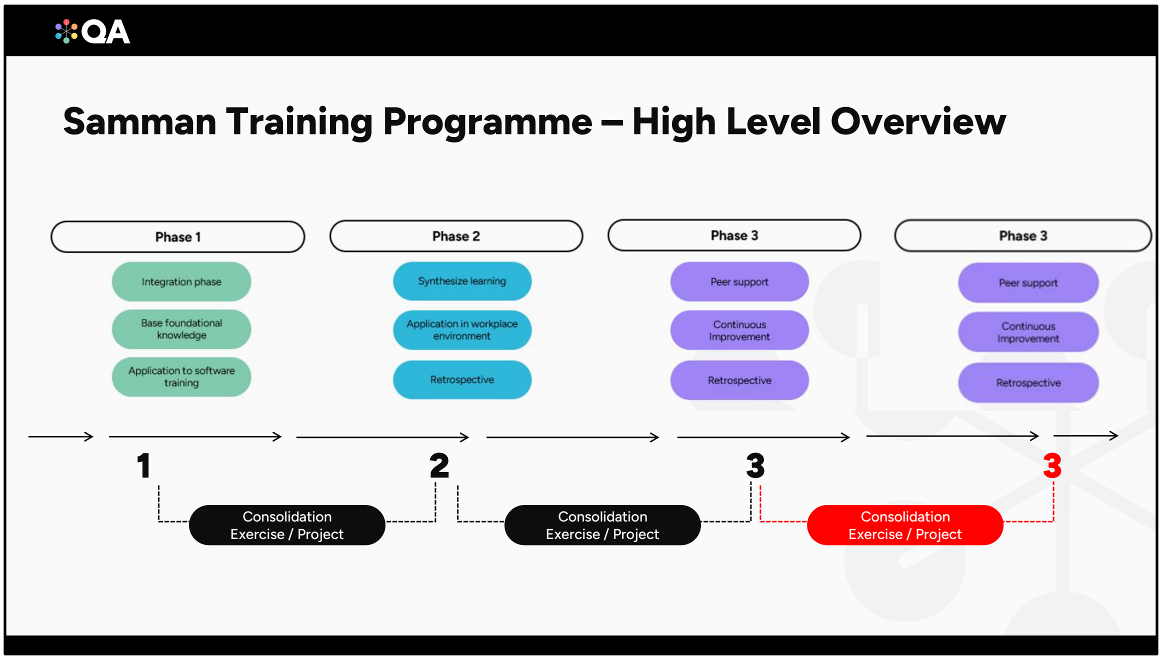
Share - Create cross-org learning and communication spaces

Peer Groups - Support coaching reflection and growth across BUs

Measure Progress/Improvements - Quantitative and Qualitative

Broadening Coaching Skills -

QA



Samman Training Programme – High Level Overview

QA
Phase 1
Integration phase
Base foundational knowledge
Application to software training
Phase 2
Synthesize learning
Application in workplace environment
Retrospective
Phase 3
Peer support
Continuous Improvement
Retrospective
Phase 3
Peer support
Continuous Improvement
Retrospective
1
2
3
3
Consolidation Exercise / Project
Consolidation Exercise / Project
Consolidation Exercise / Project

Learn. To Change.

SAMMAN TRAINING

Phase 3 - Creating a Community of Practice