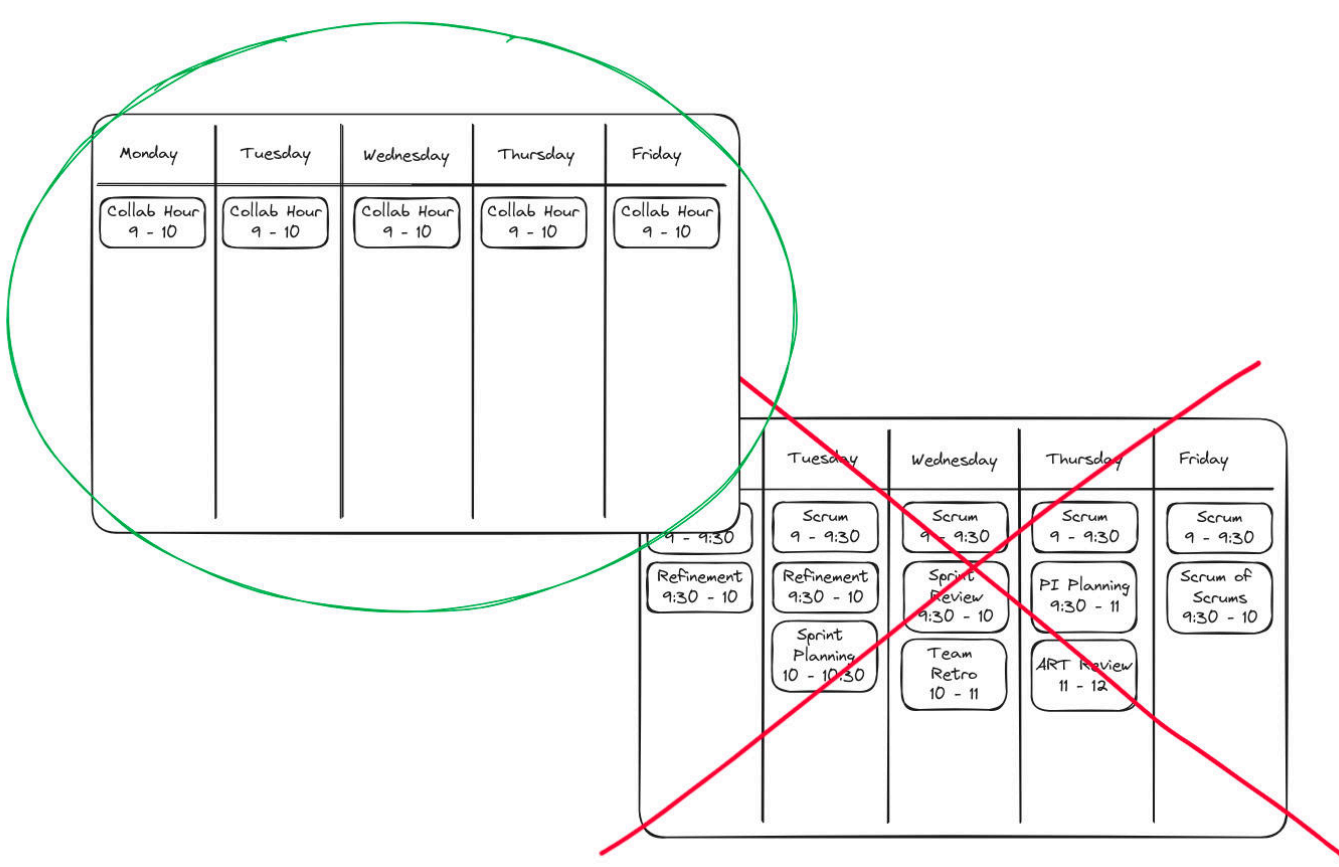


# Collab Hour

## The Complete Guide

### Summary

The model is straightforward: there's one daily meeting called “Collab Hour,” approximately one hour in duration, where all necessary discussions take place. This process *forbids* additional meetings without a clear justification. See the section on “when meetings are acceptable” for more details.



Monday	Tuesday	Wednesday	Thursday	Friday
Collab Hour 9 - 10	Collab Hour 9 - 10	Collab Hour 9 - 10	Collab Hour 9 - 10	Collab Hour 9 - 10

	Tuesday	Wednesday	Thursday	Friday
9 - 9:30	Scrum 9 - 9:30	Scrum 9 - 9:30	Scrum 9 - 9:30	Scrum 9 - 9:30
Refinement 9:30 - 10	Refinement 9:30 - 10	Sprint Review 9:30 - 10	PI Planning 9:30 - 11	Scrum of Scrums 9:30 - 10
	Sprint Planning 10 - 10:30	Team Retro 10 - 11	ART Review 11 - 12	

During each session, six descending priorities guide the discussion, though it's not necessary for all six to be completed before concluding the meeting. If you find that later priorities never get

addressed, it may be beneficial to pick up each day where you left off the previous day, but remember to avoid getting stuck in the middle of the list for several days in a row.

1. **Refine**: Refine new stories, triage bugs, and prioritize tech debt.
2. **Claim**: Ensure each team member claims some work from the top of the priority list.
3. **Unblock**: Identify blockers to progress and resolve them during this session. There's no sense noting action items if the team is capable of assisting immediately.
4. **Review**: Review all outstanding pull requests.
5. **Mob**: Identify the top priority and use mob programming to get it done. This involves the entire team including Product and QA; immediate feedback is invaluable.
6. **Retrospect**: If there's nothing urgent, review retrospective items. Each day, the team should add items to a board asynchronously, so that they can eventually be reviewed together in this meeting.

Customers should regularly should join the “Collab Hour” session to provide feedback and answer questions. This works fairly well at two weeks, which is a similar cadence to most sprints in Scrum, but the team should agree on a schedule that makes the most sense for them. It should be noted, however, that this does not require a release at the same interval. Wherever possible, favor continuous deployment with feature flags to minimize long-lived feature branches.

While no particular metrics are required by this framework, teams may choose to track data for continuous improvement. Excessive process overhead is generally discouraged, but there is only one absolute prescription regarding metrics — any performance data generated by the team must remain strictly confidential. If such data is consumed by external parties, it may lead to misconceptions which could wrongly influence how the team is handled or perceived. The team's reputation should be judged by the quality of its output alone. Therefore, only those metrics that focus on value delivery should be made available externally. Read more in the section on “[Metrics](#).”

*“Working software is the primary measure of progress.”  
— Agile Manifesto*

# The Details

---

## When meetings are acceptable

Always favor “working sessions” over “meetings.” When determining whether scheduled time is necessary, it's required to answer “yes” to at least one of these two questions:

1. Will this session accomplish a tangible outcome?
2. Will avoiding this discussion impose a blocker to progress?

Additionally, we should also answer “yes” to this one follow-up question.

1. Is this equally true for every single person in attendance?

If the answer to either of the first questions is “no,” then the meeting should not be planned. If the answer to the follow-up question is “no,” then all participants who are not required should be excused. These guidelines should be observed with strict discipline to avoid unnecessary interruptions.

If a meeting is necessary and it involves the entire team, the following day's Collab Hour session should be considered before scheduling a standalone meeting. If the topic can be addressed during the Collab Hour session, then it should be. If the topic is too urgent to wait, or it only concerns a couple members of the team, then additional time may be scheduled, but it should be kept as focused as possible.

All meetings should have a clear agenda. If the agenda does not align with the questions above, then the invited participants will be expected to decline the meeting. The facilitator should be held accountable for scheduling only necessary meetings. Productive work should always be the priority.

With that in mind, collaborative sessions are encouraged, such as pair or mob programming. With empathy, take into consideration your coworkers' social tolerance levels before allocating extra time. For some, Collab Hour alone may be pushing the limit. The important part is optimizing team synergy, for the most enjoyable and productive collaboration experience.

## Refine

Refinement is the process of preparing tasks for development. The goal is to have a clear understanding of the work required before development begins.

While it is important to prepare tasks for development, it is equally important to avoid spending the entire hour refining tasks. Focus on refining only the most important tasks for the day, and make

sure to leave time for other priorities.

Some days, you may find that there are no tasks to refine. This is perfectly normal, and in fact better for the sake of productivity. Simply move on to the next priority.

The following are some best practices for refining different types of tasks.

### Stories

Stories are the most common type of task in software. They are typically written in the form of a user story, which describes a feature from the perspective of the user: "As a...I want...so that..." This helps orient the team around the user's needs, rather than the technical implementation.

Most, if not all authors of the original Agile manifesto agree that stories should not be estimated in points nor time, so it's best to avoid this practice. Instead, use this time to determine whether a story is as small as it can possibly get while also delivering a complete feature, or "vertical slice." Use Ron Jeffries' approach to crafting user stories for best results.

When it comes to large collections of stories, Jeff Patton's "story mapping" technique is a great way to visualize the user journey and prioritize stories based on user needs.

### Bug Reports

Careful bug reporting ensures smooth transitions from QA to development. Poorly written bug reports can lead to unnecessary confusion and delay their resolution. Below is a checklist of items that should be included in every bug report.

1. **Environments:** This is not only the environment the bug was reported in, but rather *all* environments where the bug exists. (Common environments might include: DEV, TEST, QA, and STAGING.) This helps a developer understand when and how the bug was introduced.
2. **Steps to reproduce:** Provide a comprehensive list of all user actions from the moment the app is opened until the bug is observed.
3. **Scenario:** It's the responsibility of the bug reporter to outline the details of the scenario and reproduce the issue in the development environment. All the necessary prerequisite data must be in place for the developer to begin working. If reproduction is not possible, this should be noted during triage before the task is picked up.
4. **Expected vs Actual:** Clearly articulate the expected behavior versus what is observed. This reduces uncertainty and provides guidelines for acceptance to close the bug.

To triage bugs effectively, determine the severity in terms of "low," "medium," or "high." If a bug is deemed "low," it may be deferred to a later date. A "medium" severity bug should be resolved

reasonably soon, but it's not necessary to interrupt work currently in progress. If a bug has a "high" severity, all other work should be paused until it is resolved as quickly as possible.

The severity of a bug is not necessarily related to its complexity. A "high" severity bug may be a simple fix, while a "low" severity bug may be difficult to resolve. The severity is determined by the impact the bug has on the user experience.

If a bug is found with a story that is currently in progress and unreleased to production, avoid creating a separate bug report. Instead, return the story in question to the developer, as additional tracking will only create confusion. Bugs found in active stories do not need to be assigned a severity, as no users will be impacted until the story is released.

## Technical Debt

Technical debt, or just "tech debt," is the cost of additional work created by delaying long term solutions to expedite business value. Tech debt is not inherently bad, but it should be managed effectively to avoid accumulating to the point where it impedes progress.

It's also natural for unintentional tech debt to accumulate over time, as the system grows and new requirements emerge. This is understood and expected, and as such ought to be factored into the available time for development.

Introducing tech debt deliberately is primarily a choice for engineers to make, not product owners. Product owners may guide the discussion in terms of the value to be gained, but engineers possess the technical expertise to make informed decisions around acceptable risk and quality. In a similar vein, engineers should also author and track tech debt tasks, as they are best equipped to understand the work involved and its implications.

To allow the necessary time to address tech debt, engineers must be given sufficient time each week to work independently of product demands. This time should be protected and not used for additional feature work.

## Claim

All team members should have a clear understanding of their tasks for the day. If anyone is left without work to do, Collab Hour provides an opportunity to address that.

The stories should be prioritized from top to bottom, and should be claimed from the top of the list. Cherry-picking from the middle or bottom of the list is discouraged; the product owner's priorities should be respected to ensure the team is always delivering the highest value.

Some tasks, particularly complex stories, may require the collaboration of multiple engineers. A well-defined story (vertical slice) often spans across the frontend, backend, and database. In such cases, multiple individuals may claim the same story, or subtasks could be created to track each

layer independently. This often requires a high degree of collaboration, thus pair or mob programming is encouraged.

## Unblock

If any team member is blocked, whether by something outside their control or simply by a difficult task that requires assistance, then the team should work together during the Collab Hour session to resolve the issue promptly.

If the organization and team are structured properly, then no blocker should lie beyond the team's power to rectify internally. Addressing blockers in real-time eliminates the need for deferred action items and facilitates a swift return to productivity.

If the team is unable to resolve a blocker during the Collab Hour, then the team should agree on a plan to address it before the next session. This may involve escalating the issue to a different team or individual, but the goal is to ensure that the team can continue working without impediment.

## Review

While active collaboration may limit the need for reviews, there are occasions where individual work is more suitable. If any such code is awaiting approval, Collab Hour presents an occasion to make any corrections as a group, then merge once it has unanimous approval. The goal is to adjust and merge code during the group review, rather than leaving comments for the pull request owner to address at a later time.

## Mob

Mob programming is a tactic for group collaboration on a single computer. With remote work becoming more commonplace, this can be accomplished by using tools such as VSCode's Live Share extension, or IntelliJ's Code With Me. You can try different approaches and adapt it to what works best for the team, but essentially, mob programming is the idea of rotating the following roles at regular intervals, such as every ten minutes.

1. **Driver:** The typist, responsible for implementing instructions provided by the navigator. This role offers valuable learning opportunities.
2. **Navigator:** Coordinates input from team members and guides the driver. This role is ideal for leading the team.
3. **Advisors:** Everyone else in the room is tasked with open communication, answering questions, and making suggestions. This role is best for offering a specific set of knowledge or expertise.



While these roles provide structure, teams should feel free to adjust them as needed. A mature team may even spontaneously switch roles based on emerging ideas or needs.

During the Collab Hour session, the team should identify the highest priority task and use mob programming to complete it together. If the clock runs out before the task is done, finish the current thought and then commit the code before ending the session. Engineers may choose to continue working on the task after the session ends, but after Collab Hour, the team is free to drop off as needed.

If there is no urgent task to address, then the team should move on to the next priority. Remember not to neglect the retrospect portion for too many days in a row.

## Retrospect

*"At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly."*  
— Agile Manifesto

A key part of Agile is continuous improvement, often achieved through retrospective sessions. Traditionally, teams use methods like sticky notes or digital equivalents to categorize ideas into columns such as "start, stop, continue." However, this approach requires the interruption of ongoing tasks to meet and discuss.

An alternative approach is to continuously reflect throughout the day. The retro board should remain accessible at any time so ideas can be recorded as they arise. During the Collab Hour session, the team can work together to agree on process changes to address each suggestion. The intent is to take each item as far as possible before beginning a group conversation around it. This approach is a more efficient use of time and primes the discussion to be more fruitful.

## Customer Feedback

Feedback is a critical component of any successful team. It is important to regularly solicit feedback from customers to ensure that the team is building the right product, constantly adapting to change rather than focusing too much on scope and requirements up front.

## *"Responding to change over following a plan"* — Agile Manifesto

Customers should be invited to join the Collab Hour session to create an open dialogue, so that the team can better understand the needs and expectations of the end users. The frequency of customer feedback sessions should be determined by the team, but the standard is every two weeks.

These feedback sessions do not require a release at the same interval. It's often best to release features as soon as they are ready, rather than waiting for a predetermined release date.

While not specifically required, a continuous deployment strategy works well with this model. Feature flags allow the team to release features to production without making them visible to end users. This minimizes the risk of long-lived feature branches and allows the team to gather feedback early and often.

## Metrics

As stated, teams are not required to track metrics, but they may opt to do so. All performance metrics must be kept strictly confidential, for the team's eyes only, apart from management.

Metrics focused on "value delivered," such as those listed in the [DORA Metrics](#), may prove more valuable to external parties than velocity or individual performance.

Performance tracking places unnecessary stress on the team, which can lead to a decrease in morale and thus increase turnover. Furthermore, surveillance of such metrics often drives the wrong behavior, such as inflating numbers and rearranging tasks to make the burndown chart look better, rather than focusing on code quality, customer value, or business revenue.

While performance metrics may seem easy to measure, ignoring nuance in the circumstances may lead to incorrect conclusions. For example, a senior engineer may go several weeks without delivering a single story, but they may be providing invaluable mentorship to the team. The value of this mentorship is not reflected in the metrics, but the resulting increase in team productivity is undeniable.

The full and accurate truth is not easily quantifiable. This is a fact of life. To make the best decisions, we must accept this uncomfortable reality and remain cognizant of the limitations of our data. Instead of predictability, we should optimize for effectiveness, which inherently involves a



degree of uncertainty. The alternative is to optimize for metrics, which is a dangerous path that often leads to poor outcomes.