THE WORKSHOP

# Ensemble Programming

A pattern for putting the whole team on one problem at one keyboard, rotating roles on a short timer, so that knowledge spreads as the code is written and review happens at the moment of writing instead of the week after.

2026-12-11
barkingiguana.com/writing/the-workshop-ensemble-programming/

A facilitator's playbook — print, annotate, and keep on the table.

# Contents

*One person types while the rest of the team navigates. Ensemble programming, mob programming with an LLM*

*LLMA neural network trained to predict the next token in a sequence, large enough that it generalises to tasks it wasn't explicitly trained for. in the typing seat, catches problems in real time, builds shared understanding, and makes the code something the whole team owns. Worked example: The Team Navigates, the LLM Types.*

## Ensemble Programming

Ensemble programming puts the whole team on one problem at one keyboard, rotating roles on a short timer, so the code is written and reviewed simultaneously and the knowledge spreads as it's produced. Originally called mob programming, coined by Woody Zuill at Hunter Industries in the early 2010s; ensemble programming is the same practice with a friendlier label. Occasionally confused with pair programming (the two-person case) and with code review (sequential rather than simultaneous); ensemble programming is a superset of the first and a substitute for the second.

**At a glance**

- *Who, for how long:* a facilitator who doesn't code, plus three to six people who'll work in or maintain this area of the codebase (the person who knows it best, the person who knows it least, and anyone who'll be on call for it). Two hours.
- *What you walk out with:* committed, reviewed code; shared knowledge of the area across everyone in the room rather than just the author; team calibration on naming, style, and approach; a list of follow-ups with owners.
- *When to reach for it:* unfamiliar territory, knowledge that needs to spread, a new joiner learning the codebase, or a tangled legacy area where solo work produces more bugs than progress. Not for trivial well-understood work, groups larger than six (split them), or anyone who hasn't agreed to try the format.

## What's It For

A team has to integrate with a new service. Nobody has done it before. One developer volunteers, spends three days in the documentation, emerges with working code, and puts up a pull request. The reviewer glances at it, says "looks good," and merges. Two weeks later something goes wrong with the integration at 3am. The on-call engineer has never seen this code. The person who wrote it is asleep. The code works, but the knowledge lives in exactly one brain.

Or: a team has a gnarly piece of legacy code nobody wants to touch. Every change breaks something. Every bug report leads to another developer having to learn the whole tangled area. The tangle compounds because the people who've paid the learning cost don't share the map they've built.

Or: a senior developer's code goes to review and the reviewer, who is more junior, rubber-stamps it because they don't feel qualified to push back. The review is performative. The bug that should have been caught ships.

These are all the same problem, which is that writing code is a solo activity and everything that gives code its long-term value, review, knowledge transfer, design challenge, calibration across people, is supposed to happen afterwards, by people who weren't there when the decisions were made. Ensemble programming moves all of that into the moment of writing. The driver types. The navigator directs. Everyone else thinks ahead. The decisions get argued while they're being made, and the review is the act of writing.

It feels inefficient. It isn't. It trades fifteen minutes of "I could have typed this myself" for a week of "nobody else knows this code."

Reach for it when:

- The team is entering unfamiliar territory: a new language, framework, service, or problem domain
- You need knowledge of a specific area of the codebase to spread across the team
- A new team member needs to learn the codebase quickly
- The work is tricky enough that multiple perspectives make the solution better
- Code review is a bottleneck and review feedback is arriving days after the code was written
- An SRE team is writing a new runbook or instrumentation that everyone needs to understand, not just the author
- The legacy code area is so tangled that solo work on it produces more bugs than progress

## What It's Not For

Skip it when:

- The work is trivial and well-understood. One person will finish it in ten minutes.
- The team hasn't agreed to try it. Forced ensembles produce resentment, not code.
- People need deep individual focus on separate tasks for a while. Don't break that focus to perform togetherness.
- The group would be larger than six. Above that the rotation is too slow and people disengage. Split into two ensembles working in parallel on different parts of the same problem and reconcile at the breaks.

Trade-offs to know going in. The benefits are real: code is written and reviewed simultaneously so the review backlog disappears for this work; knowledge of the area spreads across everyone in the room, not just the author; the team calibrates on naming, style, and approach in ways that improve all subsequent work; new joiners learn the codebase by building it, not by reading it; design decisions get argued while they're being made, not in a pull request a week later; and the on-call engineer for this code has seen the code, rather than meeting it at 3am.

The costs are real too: two hours of three to six people on one task (real person-hours, not cheap); cognitive fatigue, because ensemble programming is harder than solo programming, measured in intensity per minute; an emotional cost of seniors relinquishing control and juniors being watched; and a first session almost always feels slow. Teams that judge it on the first session quit early and miss the payoff.

The classic failure modes: role discipline collapses and the driver gets bombarded from all sides; one person dominates as navigator and the rotation becomes a typing exercise; the task is too big, too small, or too solo-appropriate and the format fights the work; the team never does a second session, so the format never becomes comfortable.

Stop a session that's already started if:

- The first rotation ends and the team is still arguing about the rules
- Someone is visibly distressed by the format and a quiet conversation hasn't helped
- The code has gone backwards for thirty minutes and nobody can explain why

## Definitions & Background

The ensemble works through Woody Zuill's strong-style pairing rule: *for an idea to go from your head to the computer, it must go through someone else's hands.* The driver is the navigator's hands, full stop. Ideas that occur to the driver during their turn don't go in the editor; they get banked, raised when the driver speaks as an ensemble member, or brought when the rotation puts them in the navigator seat.

The roles, on this discipline:

- Driver: hands on the keyboard. Types what the navigator says. Does *not* make decisions about what to type. The navigator calibrates the level of abstraction to the driver's familiarity with the code: spell characters for a junior driver, speak in intents for a senior driver (*"extract that into a function called calculateDeliveryFee"*). When the driver disagrees with an instruction, they type it anyway, finish the current step, then voice the concern *as ensemble input* once the navigator is ready to hear it.
- Navigator: directs the driver. Decides what happens next. Speaks to the driver in instructions the driver can execute. The navigator is the only person who speaks *to* the driver.
- Ensemble: everyone else. Thinks ahead. Raises concerns. Suggests ideas. Spots bugs. Reads the tests. The ensemble speaks *to the navigator*, not to the driver. The navigator decides what to incorporate.

This discipline is what holds the ensemble together. Without it, the driver gets hit with conflicting instructions from five directions and the session collapses into chaos. The instruction *"nobody else talks to the driver"* is the one rule you enforce hardest, for the whole first session, with an apology each time.

With an LLM in the driver seat. The strong-style rule still holds, but the contract shifts. (Cursor- and Copilot-style tooling: the LLM proposes a "diff" (a code change preview) and a human "applies" it to the file.)

- The LLM stays in the driver seat across rotations; it doesn't tire, so only the navigator and ensemble seats rotate.
- One human acts as co-pilot: reads the LLM's output aloud and confirms each diff before apply. The co-pilot is the LLM's hands, ratifying intent into action, not the navigator.
- The navigator still speaks *intent* (*"extract this branch into a guard clause"*, an early-return that handles edge cases at the start of a function, e.g. *"if user is null, return"*), never prompts directly. Prompting

PromptThe input you hand to an LLM – system instructions, user message, examples, retrieved documents, tool descriptions, the lot. directly turns the navigator into a solo developer with an audience.

- The biggest LLM-specific failure mode is the silent rubber-stamp: plausible-looking code lands and the room nods. The fix is to require the navigator to *restate the intent* before any apply, so the diff is checked against what was asked for, not what the LLM produced.

The rhythm is short rotations, strict roles, continuous conversation between the navigator and the ensemble. The navigator is not working alone; they're the spokesperson for a group that's thinking ahead of them.

## Inputs

- A real task to work on, sized for the level: a self-contained kata for a learning session, a single feature/refactor/integration for a Standard session, sustained work for a Day Level team.
- A working dev environment before the session starts. The build runs, the tests pass, the editor opens. Fifteen minutes fighting the toolchain in front of the team is a guaranteed energy killer.
- A shared workstation: one keyboard, one big screen everyone can see. Remote ensembles use screen-share with hand-off control.
- A timer the whole room can see, set to the rotation length.
- A whiteboard for orientation and for the moments when the keyboard isn't the right place to think.
- The right people in the room (see *Who's Needed*) and an explicit agreement from the team to try the format.

## Outputs

What lands at the end of a session:

- Committed, reviewed code, written and reviewed at the same time, so the review backlog for this work has already cleared.
- Shared knowledge of the area across everyone who was in the room, not just the author. The on-call engineer has seen the code rather than meeting it at 3am.
- Calibration across the team on naming, style, and approach: improvements that carry into all subsequent solo work.
- A list of follow-up items named during the session: tech debt spotted, tests missing, refactors deferred, each captured with an owner.
- A short retro signal: rotation length that worked, what felt good, what felt awkward, what to try next time.

These outputs feed straight into:

- Event Storming. Event Storming builds shared mental models at the domain level; ensemble programming builds shared mental models at the code level. They address the same problem at different layers of the stack. A team that does both rarely finds itself saying *"nobody else knows this code."*

- Example Mapping. An Example Mapping session produces rules and examples that become test cases. An ensemble programming session is an excellent place to turn those tests into code. Run Example Mapping Monday, ensemble the implementation Tuesday.
- Retrospectives. Ensemble programming surfaces team dynamics (who coaches, who dominates, who freezes) faster than any other technique. A retrospective after the first few ensemble sessions is where the team names and fixes those dynamics.
- Threat Modelling. When you're writing security-sensitive code, ensemble programming and threat modelling compose naturally. Threat model the feature, then ensemble the implementation with the threats visible on a whiteboard next to the screen. Every line of code gets written in the presence of the threats it has to defend against.

## Who's Needed

Facilitator. Runs the timer, enforces the role discipline, coaches new navigators and drivers, and calls the breaks. Does not participate in the coding itself on a first session; the facilitation is a full-time job until the team knows the rhythm. After a few sessions the role can dissolve into the ensemble.

The ensemble. Everyone who will work in this area of the codebase, or who needs to learn it. This includes:

- The person who knows the most about the problem (they'll navigate for the tricky bits)
- The person who knows the least about the problem (they'll learn by navigating under coaching)
- Anyone who'll be on call for this code once it ships

For SRE work, this means the engineers who'll be paged, not just the author. For feature work, it means the developers who'll maintain the area, not just the one picking up the ticket.

Group size: 3-6. Three gives you enough perspectives for the ensemble to work. Six is the ceiling before turns come too infrequently and people disengage. Above six, split into two ensembles working in parallel on different parts of the same problem and reconcile at the breaks.

Who to leave out:

- Passive observers. "I just want to watch" distorts the dynamic and absorbs attention. Either they participate or they're not in the room.
- People who need deep individual focus right now. If someone's head is in a completely different problem, forcing them into the ensemble produces resentment. Let them work solo and catch up with the ensemble's output later.
- Managers who will interrupt. An ensemble protecting its focus is one of the most productive states a team can reach. A manager asking "quick question" breaks it every time. If a manager wants to participate, they participate by the same rules as everyone else: driver, navigator, ensemble, rotation.

## How To Run It

| Phase | Duration | Materials | Key question |
|---|---|---|---|
| Orientation | 10 min | Whiteboard | "What are we building? Where do we start?" |
| Working rotation, block 1 | 40 min | Keyboard + big screen + timer | "Navigator, what's the next small step?" |
| Break | 10 min | , | , |
| Working rotation, block 2 | 40 min | Keyboard + big screen + timer | "Navigator, what's the next small step?" |
| Commit, retro, wrap-up | 20 min | , | "What did we learn? Does the code ship?" |
| Total | 2 hours | | |

The rotation inside each 40-minute block is the thing that matters most: 4 minutes for first-timers, 5-7 minutes once the team is comfortable, 10 minutes maximum for experienced teams. The timer is non-negotiable. When it goes off, the driver moves to the ensemble, the navigator becomes the driver, and the next person in the rotation becomes the navigator.

### Phase 1. Orientation (10 min)

Before the timer starts, do three things:

Explain the roles. Even if people have done ensemble programming before, recap the rules out loud. Name the driver-navigator-ensemble roles and the one-voice-to-the-driver rule. The rules will be tested within the first rotation; make sure everyone heard them.

Set the goal at a high level. Not the whole design, just the shape:

> *"We're going to add a delivery fee calculation. We'll probably need a new function, some tests, and a hook into the order total. That's the shape; we'll figure out the details as we write."*

Pick the first driver and navigator. For a first session, put a middling-experience person in the navigator seat and a confident typist in the driver seat. The first rotation sets the tone, and you want both roles to have a fair chance of looking competent.

What to say:

> *"The roles are strict for a reason. The driver types. The navigator directs. Everyone else thinks and speaks to the navigator. Nobody speaks to the driver except the navigator. If you break that rule, I'll interrupt, and I'll do it with a smile, but I will interrupt."*

> *"We're going to start the timer in two minutes. The timer is four minutes per rotation, for the first block. When it goes off, driver goes to the ensemble, navigator becomes driver, next person becomes navigator. No exceptions for mid-sentence."*

> *"It will feel slow for the first twenty minutes. That's every first session. By the end of the first block it will click. If it doesn't click by then, we'll stop and talk about what's not working."*

What to watch for:

- Over-planning at the whiteboard. Thirty minutes of whiteboard design before touching the keyboard is a sign the team is afraid to start. Get to the keyboard within ten minutes. The design emerges in the code.
- One person explaining the whole approach. If the senior developer is delivering a lecture during orientation, the ensemble has already become a one-person show. Cut them off kindly: *"Let's start and figure out the rest as we go."*
- Environment setup failing. The dev environment must be working before phase 1. Fifteen minutes of fighting with the build tool in front of the team is a guaranteed energy killer. Fix it beforehand.

### Phase 2. Working rotation, block 1 (40 min)

Start the timer. The first rotation is the most important one of the session; the facilitator's job here is to enforce the discipline while the team learns it.

When the timer goes off, call the rotation out loud:

> *"Time. Driver to the ensemble, navigator takes the keyboard, next person takes the navigator seat. Go."*

Then restart the timer immediately. Don't wait for a natural stopping point. If the current navigator is mid-sentence, they hand off mid-sentence and the new navigator picks up. This feels wrong the first time it happens and is exactly the muscle the team needs to build: the person leaving the navigator seat has to have communicated their intent well enough that the next person can continue.

What to say:

> *"Driver, wait for the navigator. Navigator, what should the driver type next?"*

> *"Ensemble, talk to the navigator, not the driver. If you see something, tell the navigator."*

> *"Navigator, you can speak at a higher level of abstraction. The driver knows how to type a function declaration; you don't have to spell it."*

> *"Navigator, you're freezing up. Don't worry about the whole feature. What's the next small step? Just the next line. What should we write?"*

> *"Time. Handoff."*

What to watch for:

- The driver freelancing. The driver decides they know what to type and starts typing without navigator instruction. *"Driver, wait for the navigator. Navigator, what's the next step?"* Every time. The rule has to hold or the session falls apart.
- The ensemble talking to the driver. Someone calls out *"no, go back!"* directly at the driver. Interrupt: *"Talk to the navigator. They'll decide."* This happens constantly in the first rotation and fades by the third.
- The silent navigator. New navigators freeze. Coach them: *"What's the next small step? Don't try to see the whole feature. Just the next line."* If they can't unfreeze, the ensemble can whisper suggestions the navigator can relay.

- The hyperactive navigator. A navigator spelling every keystroke to a senior driver is wasting the driver's skill. *"Speak at a higher level. Instead of 't-h-e-n space curly brace,' try 'add a guard clause.'"*
- People disengaging. Someone in the ensemble is staring at the ceiling. Give them a job: *"Ensemble, while the navigator works on this function, think about what test we should write next."* The ensemble needs active tasks or it drifts.
- Rotations being skipped. *"Just let me finish this…"* No. The timer is the timer. If the thought isn't finished, the next navigator picks it up. This is where the knowledge transfer happens.
- A rotation that's clearly too short. If every handoff is mid-thought and the code is barely progressing, the rotation may be too short for this problem. At the break, consider extending to five or six minutes.

By the end of block 1, the first forty minutes should have produced something running. Not finished, running. A test that passes, a function that returns the correct thing for the simplest case, something concrete. If nothing is running, the orientation was too vague or the task was too big.

### Phase 3. Break (10 min)

A real break. Stand up, walk around, leave the room. The ensemble is cognitively demanding in a way that solo coding isn't; the continuous social presence is the thing that wears people down, not the problem itself.

Before the break, do a thirty-second check-in:

> *"How's the rotation length feeling? Too short? Too long? About right? Anything we should adjust in block 2?"*

Then stop and actually break. Don't let the break turn into a standing meeting.

### Phase 4. Working rotation, block 2 (40 min)

Same rules, same timer, same discipline. This is when the session starts to feel good: the team has warmed up, the roles are flowing, and the code is progressing at the pace the ensemble wants rather than the pace the facilitator enforces.

What to watch for in block 2:

- Energy drops in the last ten minutes. Normal. Shorten rotations slightly if you need to, or aim for a clean commit point before the timer runs out.
- The "just one more thing" trap. The team is almost done and wants to skip the commit/retro phase to finish. Don't let them. A clean stop is part of the discipline.
- Solo drift. Someone reaches for their own laptop to "just try something quickly." *"If it's worth trying, it's worth trying together. Show us on the shared screen."*

### Phase 5. Commit, retro, wrap-up (20 min)

Stop coding when the timer ends the second block. If there's a clean commit point, commit. If there isn't, stash on a branch with a clear "ensemble WIP" label and note where the next session should pick up.

Then run a short retro. Not a full retrospective: a small one, focused on the format, not the code.

What to say:

> *"One sentence each. What felt good about the ensemble? What felt awkward? Was the code better than one person would have written alone?"*

> *"Rotation length. Four minutes? Five? What should we try next time?"*

> *"Did anyone feel left out? If yes, how do we fix that for next session?"*

What to watch for:

- "It was slow." Acknowledge honestly. Then redirect: *"Was the code better? Did we skip a review? How much rework do we expect?"* Ensemble programming trades typing speed for understanding speed. The trade is usually worth it; teams feel the loss before they feel the gain.
- "I hated being the driver." First-session discomfort. Reassure: *"It gets easier. The hardest rotation is your first one."*
- "I didn't feel like I contributed." Either the rotation was too long for the group size, or the ensemble wasn't given thinking tasks. Both are facilitator-fixable for next time.

### Worked example

See Ensemble Programming: The Team Navigates the LLM Types: the Greenbox team hitting an LLM integration they don't understand, putting everyone on one keyboard, and discovering that the four-minute rotation turns uncertainty into progress faster than a week of solo work could have. The moment the most junior person in the room navigates a senior driver through something none of them have seen before is the moment the pattern earns its cost.

## What Can Go Wrong

The expert takeover. The senior developer navigates perfectly, the rotations become a typing exercise for everyone else, and the ensemble is quiet. *Recovery:* Put the senior developer in the driver seat and a junior in the navigator seat. The senior can only type; they can contribute ideas as part of the ensemble, but the navigator decides what to do with them. This inverts the power gradient and forces the senior to coach instead of dictate. *Stop if:* The senior developer can't hold the driver role: keeps ignoring the navigator, keeps correcting from the keyboard. They need a conversation one-on-one, not another ensemble session.

The architecture argument. Two people disagree about the design approach and twenty minutes have gone by without code changing. *Recovery:* Time-box it. *"We have two minutes to decide. If we can't agree, we try approach A for the next fifteen minutes. If it's not working, we switch to B."* The ensemble can test an approach faster than it can debate one. *Stop if:* The disagreement is fundamental: the two people are actually solving different problems. Pause the session, whiteboard the question, and decide what to build *before* returning to the ensemble.

The death spiral. The code isn't working, the error messages are confusing, the energy is dropping, and every rotation makes it worse. *Recovery:* Pause the timer. Step away from the keyboard. Go to a whiteboard: *"What are we actually trying to do? Let's draw it."* Reorient, then come back. A ten-minute whiteboard break saves the next hour. *Stop if:* The problem is genuinely outside the team's depth and nobody in the ensemble knows enough to navigate it. End the session, have one person explore it solo for a few hours, and come back to the ensemble once there's a foothold.

The too-big task. Halfway through, the team realises the task is much larger than a two-hour ensemble can finish. *Recovery:* Narrow the scope. *"What's the smallest piece of this we can finish and commit today? Let's do that and pick up the rest in the next session."* Partial delivery is better than broken delivery. *Stop if:* The team can't agree on a narrower scope. The task isn't ready for the ensemble; it needs design work first.

The solo coder. Someone keeps reaching for their own laptop to "just check something quickly." *Recovery: "If it's worth checking, check it together. Show us on the shared screen."* Be firm and kind; it's an instinct, not malice. *Stop if:* They can't hold it. Let them work solo for the rest of the session and note that ensemble isn't for them right now. Don't force compliance; force it and you get resentment that poisons the next session too.

The rotation the team wants to skip. Everyone is in flow and doesn't want to break it. *Recovery:* Enforce the rotation anyway, once. Flow is good, but the discipline is what's producing the knowledge spread; breaking it once breaks it often. If the team's case is really strong, adjust the rotation length at the next break, not mid-block. *Stop if:* The team has decided collectively that rotations don't work for this session. That's a signal you're running pair programming with spectators, not an ensemble. End the session as an ensemble and let the pair continue separately.

The hardest moment in ensemble programming is when the navigator says something the driver disagrees with and the driver types it anyway. That moment feels wrong; it feels like typing someone else's mistake. But that's the technique working. The driver is practising trust; the navigator is practising communication; and more often than either expects, the approach turns out to be better than what the driver would have done alone. The whole team levels up. The code is the artifact; the team is the product.

## Next Steps

The session ends; the work begins.

Same day, the facilitator:

- Make sure the code is committed and pushed. Ensemble code that sits on a laptop defeats the purpose.
- Write a short summary: *"Here's what we built. Here's the rotation length that worked. Here's what we'd change next time."*
- If the team named any follow-up work during the session (tech debt spotted, tests missing, refactors deferred) capture each item with an owner.

This week, the product owner:

- Book the next session. Ensemble programming improves dramatically with practice. One session is an experiment; three is a habit. Commit to at least three sessions before judging the format.
- Decide what kind of work goes to the ensemble. Not everything should. A good rule of thumb: tricky, unfamiliar, tangled, or critical work to the ensemble; routine work solo. Let the team negotiate the line; the owner's job is to make the choice visible, not to make it for them.

- Watch for rework on the ensemble's output. The test of a good ensemble session is whether the code holds up without rework. If it does, the ensemble earned its cost. If it doesn't, look at what was missing: was the task too big, the orientation too vague, the rotation too short?
- Protect the ensemble from interruption. A team that ensembles gets interrupted by managers and stakeholders and helpful colleagues more than a team coding solo. Defend the focus time explicitly. *"They're in an ensemble until 4pm. Message me, I'll handle it."*

Ongoing, the team:

- Rotate the facilitator role. Once the team knows the rhythm, facilitation can rotate like the driver and navigator roles.
- Find the team's natural rhythm. Some teams ensemble for two hours a day. Some ensemble all day on tricky work and solo on routine. There's no single correct answer; let the team discover theirs.
- Keep the timer visible. The rotation discipline is what makes the format work, and the timer is what makes the discipline automatic.

## Variants

The level changes the duration and the kind of work.

| Level | Scope | Duration | Output |
|---|---|---|---|
| Kata Level *(zoom in)* | A small, self-contained exercise for the team to learn the format | 60-90 min | Practice, not shipped code |
| Standard *(default)* | One real feature, refactor, or integration | 2 hours | Committed, reviewed code + shared knowledge |
| Day Level *(zoom out)* | A team that ensembles for half or all of each day | Half day or more | Sustained flow, team calibration, continuous knowledge spread |

Kata Level (zoom in). A small, self-contained exercise (a coding kata, a refactoring puzzle, a TDD exercise) run for 60 to 90 minutes. The point is practising the format itself: the rotation, the strong-style rule, the navigator-to-ensemble conversation. Output is muscle memory, not shipped code. Run one or two of these before attempting a Standard session on real work.

Standard (default). One real feature, refactor, or integration; two hours; three to six people. Committed, reviewed code plus shared knowledge of the area. This is what most teams should start with after a Kata or two, and what the rest of this post describes.

Day Level (zoom out). A team that ensembles for half or all of each day. Sustained flow, team calibration, continuous knowledge spread. Don't start here; the discipline of the rotation takes practice to enjoy, and a first-session Day Level ensemble produces exhausted people and abandoned work. Some teams do half a day of ensemble and half a day of solo work; some ensemble the whole day. Earn it with Standard sessions first.

With an LLM in the driver seat. The rotation collapses to navigator and ensemble seats only; the LLM doesn't tire and stays as driver across the whole session. One human acts as co-pilot: reads the LLM's output aloud and confirms each diff before apply. Navigator still speaks intent, never prompts directly. The biggest LLM-specific failure mode is the silent rubber-stamp; the fix is to require the navigator to restate the intent before any apply. See the *Definitions & Background* section for the full contract.

Remote. Everyone on a video call, one person sharing their screen, hand-off control between drivers explicitly at each rotation. Slightly slower than in-person (typing latency, audio overlap, the loss of body language for "wait, hold on" signals) but the structure transfers. Use a shared timer everyone can see and a chat channel for the ensemble to drop links and code references the navigator might want.

Cross-team ensemble. Two teams who own adjacent areas of the codebase ensemble together for two hours when they're working on the integration between them. The output is the integration plus calibration on the boundary: which side owns what, which contracts each side trusts. Particularly valuable when an SRE team and a product team meet in the middle on observability or on-call concerns.

## About this playbook

This playbook is part of *The Workshop*, a reference series of facilitator playbooks published at barkingiguana.com. The canonical, up-to-date version lives at barkingiguana.com/writing/the-workshop-ensemble-programming/.

*These posts are LLM-aided. Backbone, original writing, and structure by Craig. Research and editing by Craig + LLM. Proof-reading by Craig.*