

The agents write the code. Your team decides what gets built.

Ivan Blažević · Rubycode · 30 September 2026 · 12 min read

`plan_driven` is a Ruby gem that runs a Rails team's delivery process from the terminal. An interview becomes an implementation plan grounded in your real code. You approve it, it becomes tickets, and each ticket goes to its own AI agent, which opens a pull request. Nothing merges until you've read it and every acceptance criterion has a passing test. We used it to build one feature in a new Rails app, and recorded all of it.

Coding agents got good this year. Give one a clear task and a codebase and it will write the migration, the model, the controller, the views and the specs, and open a pull request before your coffee is cold. At Rubycode we use them every day.

What didn't get better is everything around the code. Which feature, exactly? What happens when the event is full, or the organizer tries to RSVP to their own event? How do you split it so that each pull request can be reviewed in one sitting? Who agreed to the plan, and where is it written down? When the agent says it's done, how do you know every requirement has a test, and not just the easy ones?

Those questions used to be answered by a process: a plan in Confluence, a review, tickets in Jira, pull requests linked to tickets, and QA ticking off acceptance criteria. Agents made the code fast, so the process is now the slow part, and on most teams it quietly disappears. The agent gets a one-line prompt, and the decisions a team would have made together get made by the model, in a tool call nobody reads.

We didn't want to give up the speed or the process, so we wrote the process down as code.

A plan first, then one agent per ticket

`plan_driven` adds a command, `plan-driven`, to your Rails app. It takes a feature through the same phases a careful team already uses, and it won't let a phase start before the one before it is approved:

1. **Interview.** You answer the questions only people can answer: what, why, where, who, when, background and what's out of scope.
2. **Plan.** A model drafts the rest of the implementation plan from your answers, your schema and your code: the existing data structure, architectural and database changes, risks, security, monitoring and testing. You read it as a PDF, change what's wrong and approve it.
3. **Tickets.** The approved plan is split into tickets with acceptance criteria, estimates and dependencies. You steer them, approve them, and they become GitHub issues.

4. **Development.** Each ready ticket goes to its own Cursor cloud agent, which opens a pull request. A ticket starts only when the tickets it depends on are merged.
5. **Review.** You read each pull request, run the review checks, send feedback to the same agent if needed, and approve and merge from the terminal.
6. **Evidence.** Every acceptance criterion is mapped to the Cucumber scenario that proves it, and a delivery report records what was built, by whom, and who approved it.

The model does the writing. The rules are Ruby. That split is the whole idea, so we'll come back to it.

Or click through it in the browser

New in 0.2.0 is a browser wizard, mounted at `/plan_driven` in development. It has five steps: Plan, Approve, Tickets, Agents & PRs, and Proof. It doesn't reimplement anything. Every button runs one `plan-driven` command, and a panel beside the form shows that command and its output as it runs. The guards are the same, the audit trail is the same, and anything you click you could also type.

We delivered a second feature in Gather with it, **comments on events**: six tickets, six agents, six pull requests, one round of feedback, and 42 of 42 acceptance criteria proven by a passing scenario.

RUBYCODE · OPEN SOURCE GEM

plan_driven

From an idea to merged, tested pull requests, in the browser. Every click runs a real `plan-driven` command.

 **Watch the demo**
The browser wizard · one feature, start to finish · 12 min

Rubycode · plan_driven

terminal - what the wizard runs

```
$ bin/plan-driven status PD-3 --yes
```

#	Title	Kind	Pts	Status	PR
T1	Migration: Create comments table	migration	2	merged	#40
T2	Add Comment model and delete comments w...	code	3	merged	#41
T3	Show comments on the event page	code	3	merged	#43
T4	Post a comment on an event	code	3	merged	#44
T5	Let authors delete their own comments	code	2	merged	#45
T6	Let organizers delete any comment on th...	code	2	merged	#45

every ticket merged; 'plan-driven' writes the delivery report

PD-3: Comments on events (delivery report)

Status: delivered · Planets 2 · Created by Ivan Stokich · view@rubycode.com · 30 September 2020

Summary
6 of 6 tasks merged, 42 of 42 acceptance criteria are proven by 3 passing scenarios

Tickets and pull requests

#	Ticket	Status	Pull Request	Merge	Approved by
T1	Migration: Create comments table	merged	#40	2020-09-30	Ivan Stokich
T2	Add Comment model and delete comments with their event id attribute	merged	#41	#4000	Ivan Stokich
T3	Show comments on the event page	merged	#43	#4001	Ivan Stokich
T4	Post a comment on an event	merged	#44	#4002	Ivan Stokich
T5	Let authors delete their own comments	merged	#45	#4003	Ivan Stokich
T6	Let organizers delete any comment on their events	merged	#46	#4004	Ivan Stokich

Acceptance criteria and proof

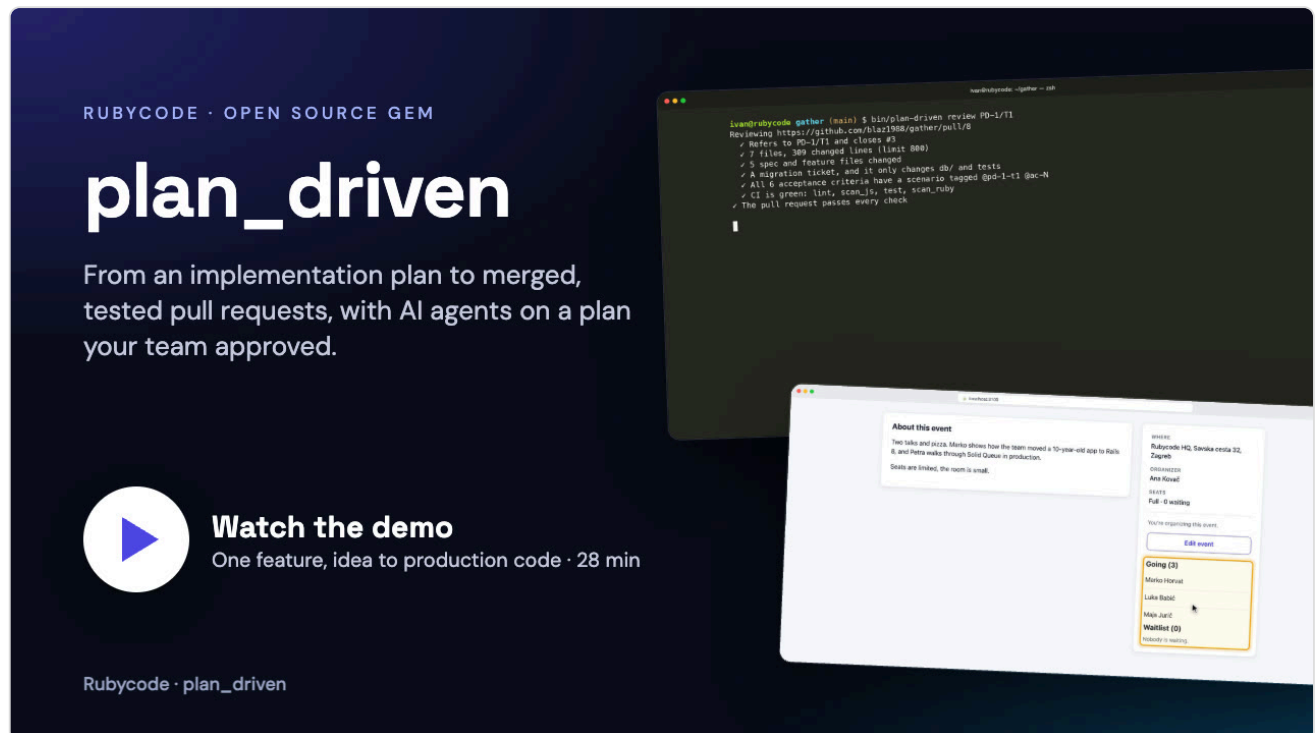
The wizard, 12 minutes, narrated: the interview as a form, redrafting one section of the plan, the tickets, six pull requests, feedback to one agent, the evidence and the feature in the app. After every click, the terminal panel shows the command that ran.

One feature, from idea to production code

To show it honestly, we built a new Rails 8 app for it, **Gather**. It's a small events site for the Zagreb tech community: you can browse events, and organizers can create them. Events have a capacity, but nothing counts seats, so organizers collect names in chat and the small rooms overflow.

The feature was **RSVPs with a waitlist**. Signed-in people RSVP and cancel, the event page shows seats left, a full event puts you on a waitlist, a cancellation promotes the first person waiting, and the organizer sees who's going. It's small enough to follow in one sitting and has enough edge cases to be interesting: two people racing for the last seat, the organizer's own event, what "cancel" does to the waitlist.

The planner and all five agents ran on Claude Opus 5.5, through a Cursor account. Here is the whole run, recorded as it happened. Nothing is staged: the plan, the tickets, the pull requests and the report are all in the Gather repository.



The screenshot shows a video player interface for a demo titled "plan_driven". The video content includes a terminal window showing a CLI command and its output, and a web application interface for an event. The terminal output shows a review of a pull request, listing changes and checks. The web application shows an event page with a "Going" list and a "Waitlist" list.

RUBYCODE · OPEN SOURCE GEM

plan_driven

From an implementation plan to merged, tested pull requests, with AI agents on a plan your team approved.

[Watch the demo](#)
One feature, idea to production code · 28 min

Rubycoder · plan_driven

```
ls@rubycoder: gather (main) $ bin/plan-driven review PD-1/11
Reviewing https://github.com/blaz1988/gather/pull/8
✓ Refers to PD-1/11 and closes #8
✓ 7 files, 349 changed lines (limit 800)
✓ 5 spec and feature files changed
✓ A migration ticket, and it only changes db/ and tests
✓ All 6 acceptance criteria have a scenario tagged @pd-1-11 @ac-N
✓ CI is green: lint, scan_js, test, scan_ruby
✓ The pull request passes every check
```

About this event

You're organizing this event.

Going (2)

- Maria Hovari
- Luke Boud

Waitlist (0)

Recently in waiting

The CLI deep dive, the whole run in a terminal, 28 minutes, narrated: the interview, reading the plan, changing a section of it, the tickets, five pull requests read and merged, two rounds of feedback, the evidence and the finished feature.

An interview, not a prompt

`plan-driven new "RSVPs with a waitlist"` asks seven questions. I typed nine lines in total, and none of them mentioned tables, models or controllers. Those are the model's job.

The draft came back with eight assumptions listed under the plan: cancelling deletes the RSVP rather than marking it cancelled, the waitlist is first come, first served, the organizer sees names but not email addresses, and so on. That list is the first thing to read, because it's where the model made a decision on your behalf.

```
ivan@rubycoder: ~/gather -- zsh

Background: Links, PRDs, existing tickets, earlier decisions (optional)
(finish with an empty line)
> Rails 8 authentication is in place; Current.user is the signed-in person.
>
Out of Scope: What is explicitly out of scope? (optional)
(finish with an empty line)
> Email notifications, paid tickets, guests (+1), and changing capacity after people have RSVPed.
>

Drafting with cursor/claude-opus-5-5...
✓ PD-1 drafted (1 attempt)
  assumed: Production runs on SQLite (per `config/database.yml`), and Rails 8.1's default `IMMEDIATE` transaction mode f
or SQLite is active. It is not overridden in `database.yml`.
  assumed: Cancelling deletes the `rsvps` row, so there is no cancelled status. Someone who RSVPs again joins the back o
f the waitlist.
  assumed: The waitlist is first come, first served by `created_at`, then `id`. Promotion is automatic and immediate, wi
th no confirmation step.
  assumed: There is no policy library (such as Pundit) in the app. Authorization stays as `Event#organized_by?` plus sco
ping through `Current.user`.
  assumed: The organizer's attendee list shows `users.name` only. `users.email_address` is not shown.
  assumed: Existing events start with zero RSVPs. Names collected in chat are not imported.
  assumed: No feature-flag system exists in the repository, so the feature is released by deploying the UI.
  assumed: The events index keeps showing total `capacity`. Seats left appear only on `/events/:id`.
✓ All checks passed
docs/plans/pd-1-rsvps-with-a-waitlist/plan.md
docs/plans/pd-1-rsvps-with-a-waitlist/plan.html
docs/plans/pd-1-rsvps-with-a-waitlist/plan.pdf
Next: read PD-1 ('plan-driven show PD-1' or the PDF), then 'plan-driven submit PD-1'.
```

Figure 1. Seven questions, then a draft. Every assumption the model made is listed, so you know what to check first.

Reading the plan

The plan is written to `docs/plans/pd-1-rsvps-with-a-waitlist/` as Markdown, HTML and PDF. It reads like one an engineer on your team would write, because it is grounded in the code: it cites

`Event#organized_by?`, the `require_organizer` filter in the events controller, and the fact that production runs on SQLite, so a race for the last seat needs an `IMMEDIATE` transaction. Before I saw it, a guard had already checked that every model, file and column in Existing Data Structure really exists.

gather/docs/plans/pd-1-rsvps-with-a-waitlist/plan.html

Background

Rails 8 authentication is in place; Current.user is the signed-in person.

Existing Data Structure

Nothing in the schema stores RSVPs, attendance or seat usage today. `events.capacity` is only displayed. The sidebar in `app/views/events/show.html.erb` renders it as `pluralize(event.capacity, "seat")`, and `app/views/events/index.html.erb` shows it on the listing.

Event — `app/models/event.rb` (table `events`)

- Columns: `id` integer not null, `title` string not null, `description` text, `starts_at` datetime not null, `venue` string not null, `capacity` integer not null, `organizer_id` integer not null, `created_at` / `updated_at` datetime not null.
- Indexes: `index_events_on_organizer_id`, `index_events_on_starts_at`. Foreign key `organizer_id` → `users.id`.
- Associations: `belongs_to :organizer, class_name: "User"`.
- Validations: presence of `title`, `starts_at`, `venue`; `capacity` must be an integer greater than 0.
- Behaviour: scope `:upcoming` (`starts_at >= now`, ordered by `starts_at`) and `organized_by?(user)`, which is the only authorization check in the app today.
- Controller: `app/controllers/events_controller.rb`. `index` and `show` allow unauthenticated access and call `resume_session`. `edit` and `update` go through `require_organizer`. There is no `destroy` route (`resources :events, except: :destroy`).

User — `app/models/user.rb` (table `users`)

- Columns: `id` integer not null, `email_address` string not null (unique index), `password_digest` string not null, `name` string not null (default ''), `created_at` / `updated_at` datetime not null.
- Associations: `has_many :sessions, dependent: :destroy`; `has_many :organized_events, class_name: "Event", foreign_key: :organizer_id, inverse_of: :organizer, dependent: :destroy`.

Session — `app/models/session.rb` (table `sessions`)

Figure 2. The plan follows the implementation plan template many teams keep in Confluence, and cites only models, files and columns that exist.

The model had a list of outstanding questions, and they were the right ones. Can the organizer RSVP to their own event? When does RSVP close? Should the events index show seats left too? Those are product decisions, and the model rightly didn't make them. I answered them in one sentence each, and `plan-driven`

redraft rewrote that section with a Decided list, down to which method enforces each decision.

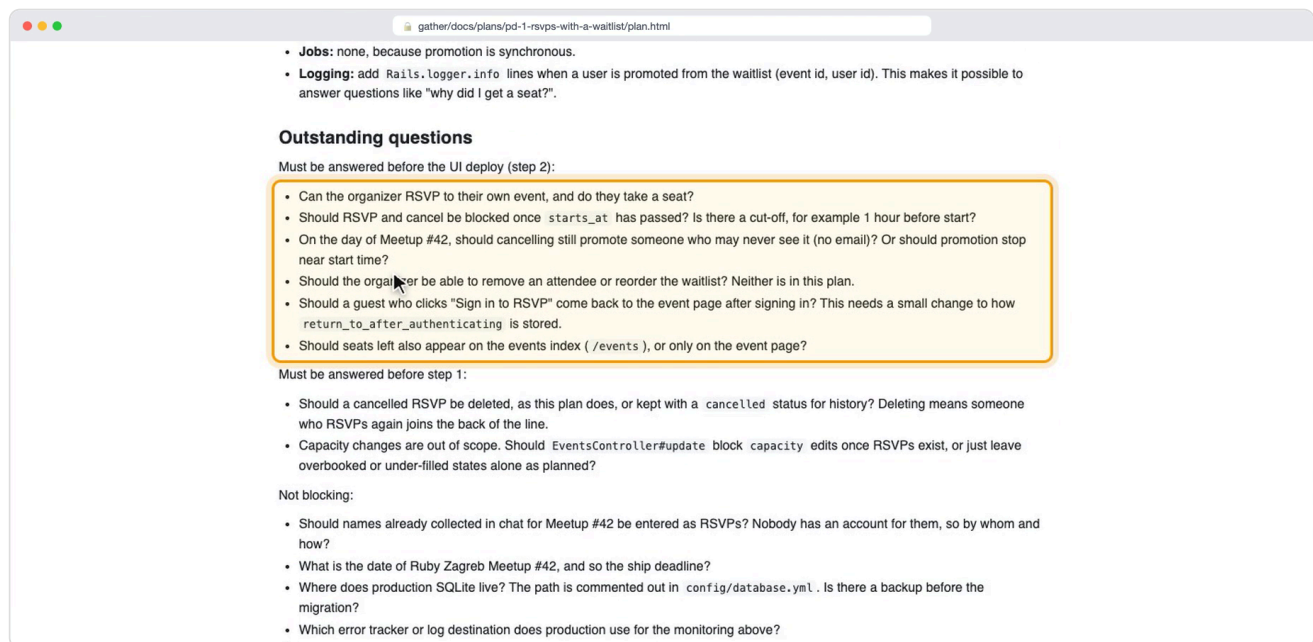


Figure 3. My answers, recorded under Decided. The questions that don't block the work stay open.

Outstanding questions is only one section. **Every section of the plan can be changed:** What, Why, Database changes, Application changes, Risks, Testing. The draft is the model's proposal, and the team has the final say. To show it, I drafted a second plan for Gather, PD-2, event categories. The model proposed a `category` string column on `events`. We'd rather have a table, so `plan-driven redraft PD-2 database_changes` "Use a categories table instead of a string column..." rewrote that one section: a new `categories` table, seeded with four rows, and a `category_id` reference on events, still in expand and contract steps. For a small change I didn't need the model at all. `plan-driven edit PD-2 database_changes` opened the section in vim, and I added a `color` column by hand.

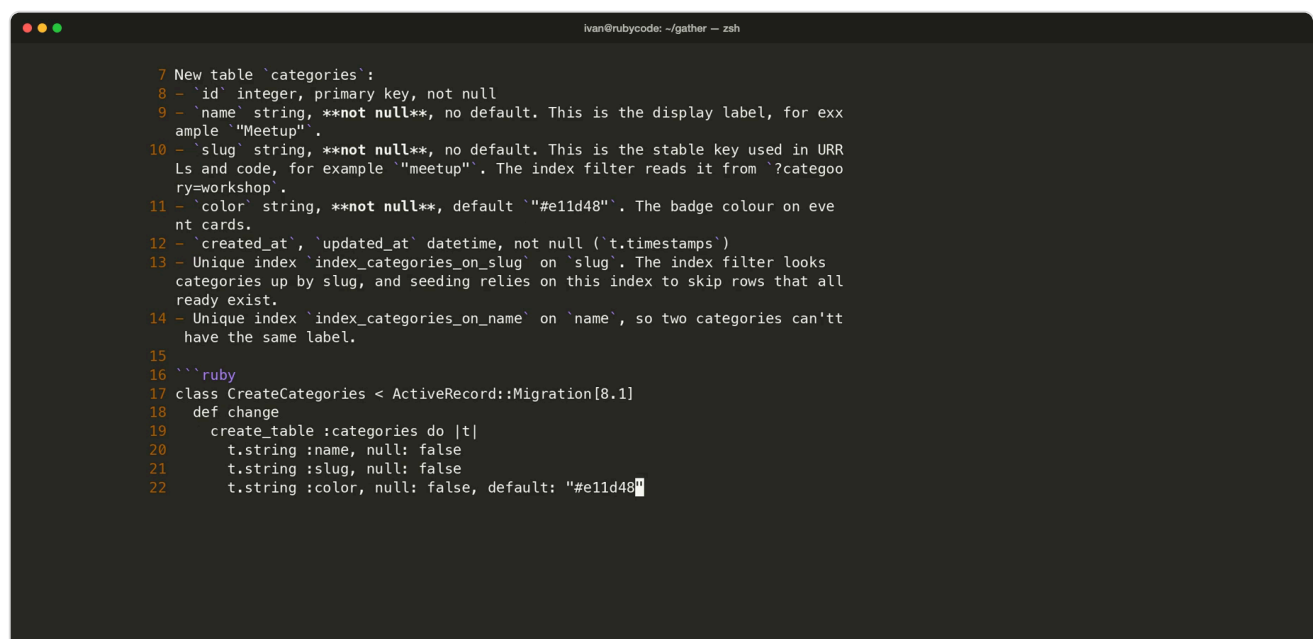


Figure 4. `edit` opens any section in your own editor. Saving makes a new revision, and the guards run again.

Every change is a new revision in the log, the plan goes back to draft, and the guards check it the same way they check the model's draft. Approvals belong to a revision, so a changed plan is approved again. Once tickets are drafted, the plan is locked and changes go into a follow-up plan.

Then `submit`, which runs the guards again, and `approve`. Revision 3 was approved. If anyone edits the plan now, it needs approving again.

Tickets you can steer

The first breakdown had nine tickets and one of them had ten acceptance criteria. A guard flagged it. The model had split the work by user action, while I wanted it split by layer so that each pull request builds on the last one. `plan-driven tickets PD-1 "Make it five tickets..."` redrafted them: the migration, the model rules, the RSVP card, the organizer's list, and seats left on the index.

The screenshot shows a web browser window with the address bar displaying `gather/docs/plans/pd-1-rsvps-with-a-waitlist/plan.html`. The document content is as follows:

Out of Scope
Email notifications, paid tickets, guests (+1), and changing capacity after people have RSVPed.

Work items

#	Title	Type	Kind	Estimate	Depends	Issue
T1	Migration: Create rsvps table	TASK	migration	1	-	-
T2	Add Rsvp model and Event rules for RSVP, cancel, waitlist and promotion	TASK	code	5	T1	-
T3	Let signed-in people RSVP, join the waitlist and cancel from the event page	STORY	code	5	T2	-
T4	Show the organizer who is going and who is waiting	STORY	code	3	T3	-
T5	Show seats left on the events index	STORY	code	2	T2	-

Estimated total: 16 points

T1. Migration: Create rsvps table
Add the new rsvps table, which links one user to one event with a status of 'going' or 'waitlisted'. This is the only schema change for PD-1. It only adds a table, so it can ship before any code reads or writes it. No model, route or UI is part of this ticket. Columns: id; event_id integer not null with a foreign key to events; user_id integer not null with a foreign key to users; status string not null with no default; created_at and updated_at datetime not null. Add a unique index on [event_id, user_id], a composite index on [event_id, status, created_at], the user_id index that t.references creates, and a check constraint named rsvps_status_check with the condition status IN ('going', 'waitlisted'). The foreign keys must not use ON DELETE CASCADE, because deletes go through model dependent: options so that callbacks run.

ACCEPTANCE CRITERIA

- Running `bin/rails db:migrate` creates the rsvps table with event_id, user_id, status, created_at and updated_at, all NOT NULL, and status has no default
- Inserting a second row with the same event_id and user_id fails with `ActiveRecord::RecordNotUnique` through the index `index_rsvps_on_event_id_and_user_id`

Figure 5. The tickets are added to the plan's PDF, with their dependencies, estimates and acceptance criteria, so they're read where the plan is read.

`approve-tickets` created a GitHub issue for each one, with the story, the acceptance criteria and the implementation notes straight from the plan. That issue is what the agent works from, and what I'll check its pull request against.

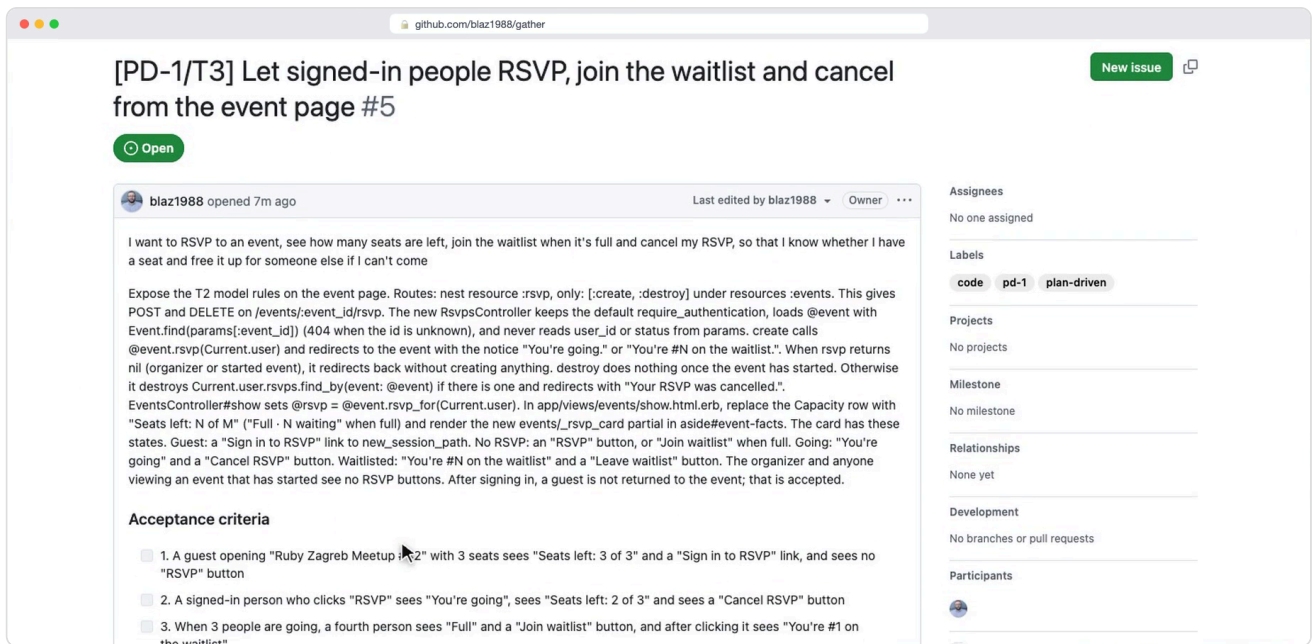


Figure 6. Each issue carries the plan key and the ticket, so anyone can trace it back to the approved plan.

One agent per ticket, in order

plan-driven develop PD-1 started one agent, for T1, because the other tickets depend on it. Each agent gets the ticket, the parts of the plan it needs, our team's conventions, and the rules its pull request will be checked against: which Cucumber file to write, how to tag each scenario, what it may and may not change. plan-driven prompt PD-1/T1 shows the exact text, so there is no hidden prompt.

Reading the pull request

When it finished, the agent opened **pull request #8**. I read it on GitHub, like any other pull request: the migration with its foreign keys, the unique index and the status check constraint, the regenerated schema, and a Cucumber feature with one scenario per acceptance criterion.

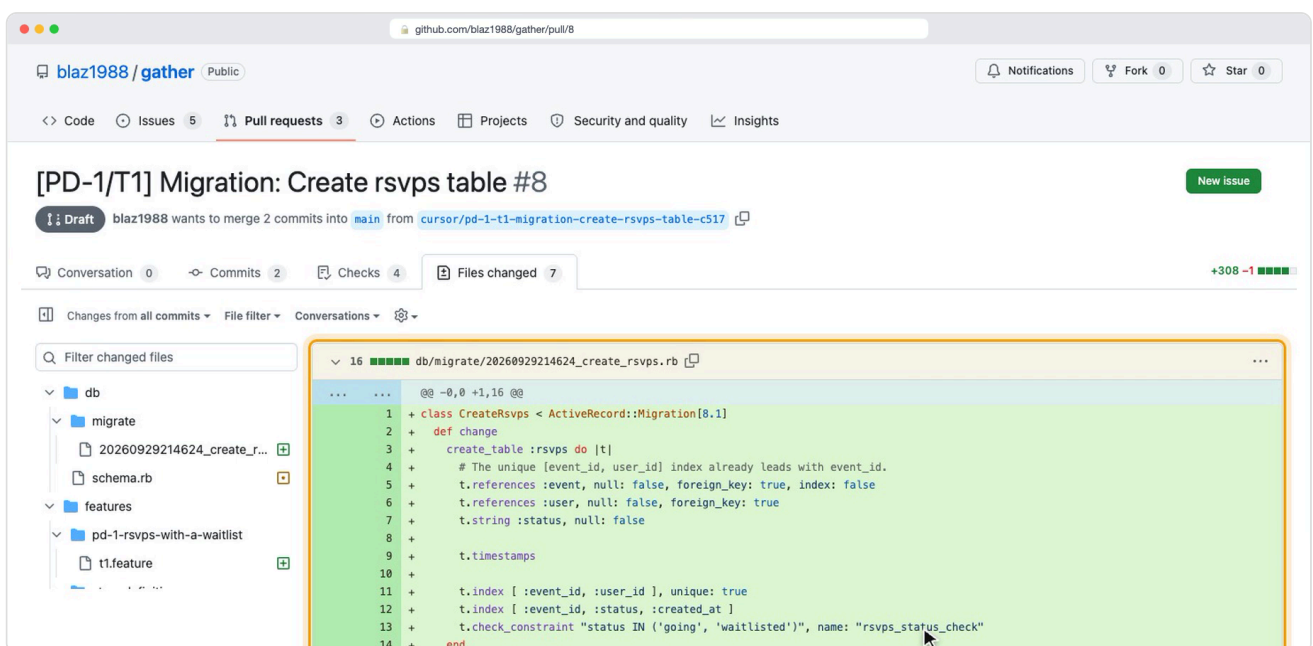


Figure 7. The migration ticket's pull request: the schema change, and a scenario for each acceptance criterion.

Then `plan-driven review PD-1/T1` checked what a reviewer shouldn't have to check by hand.

```
ivan@rubycode: gather (main) $ bin/plan-driven review PD-1/T1
Reviewing https://github.com/blaz1988/gather/pull/8
✓ Refers to PD-1/T1 and closes #3
✓ 7 files, 309 changed lines (limit 800)
✓ 5 spec and feature files changed
✓ A migration ticket, and it only changes db/ and tests
✓ All 6 acceptance criteria have a scenario tagged @pd-1-t1 @ac-N
✓ CI is green: lint, scan_js, test, scan_ruby
✓ The pull request passes every check
```

Figure 8. The review checks: the ticket and issue, the size, the specs, that a migration ticket only changes the database and tests, a scenario for every criterion, and green CI.

`approve-pr` records my approval and posts it on the pull request. `merge` asks me to type the ticket key, merges, and says what's ready next. For the pull requests with UI, I checked out the branch and clicked through it before approving, because the checks prove the criteria have tests, and only a person decides whether the feature is right.

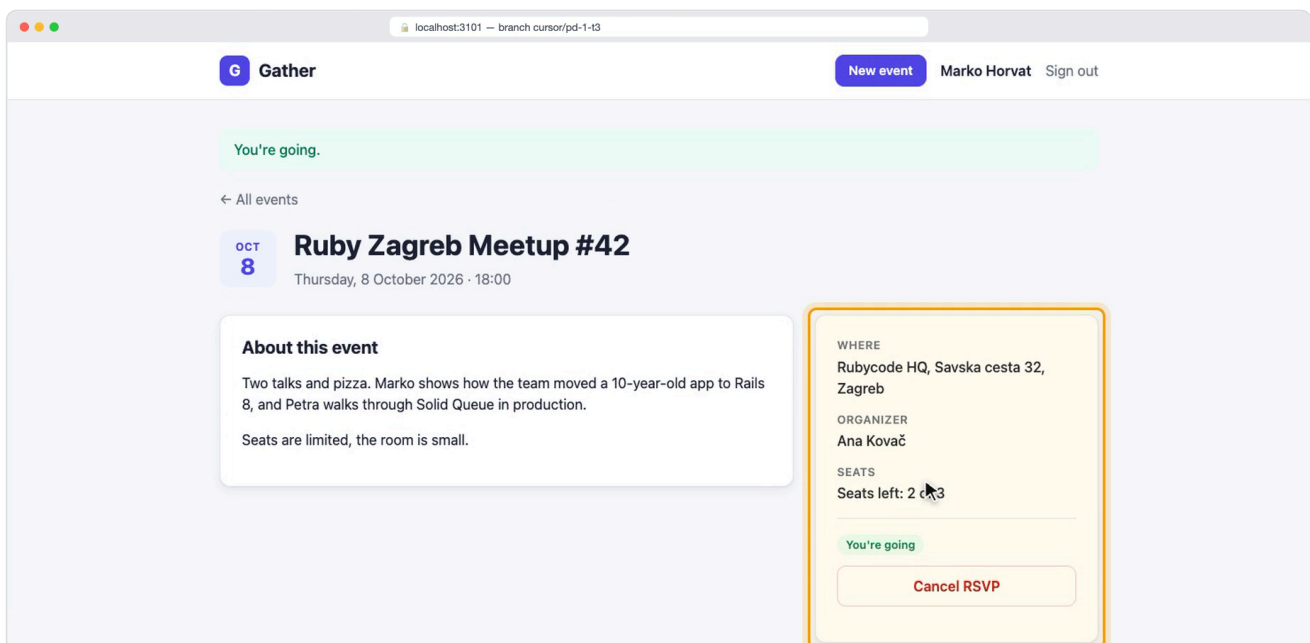


Figure 9. The RSVP card on the T3 branch, tried locally before it was approved.

Feedback goes back to the same agent

Two pull requests needed another round. T5, seats left on the index, was opened before T3 was merged, so its CI had run without T3's code. The review caught that the branch was two commits behind main. The helper also repeated a rule that already lived in `Event#seats_left`. One `plan-driven feedback` command sent both points to the same agent, which pushed to the same pull request, and the next review was clean.

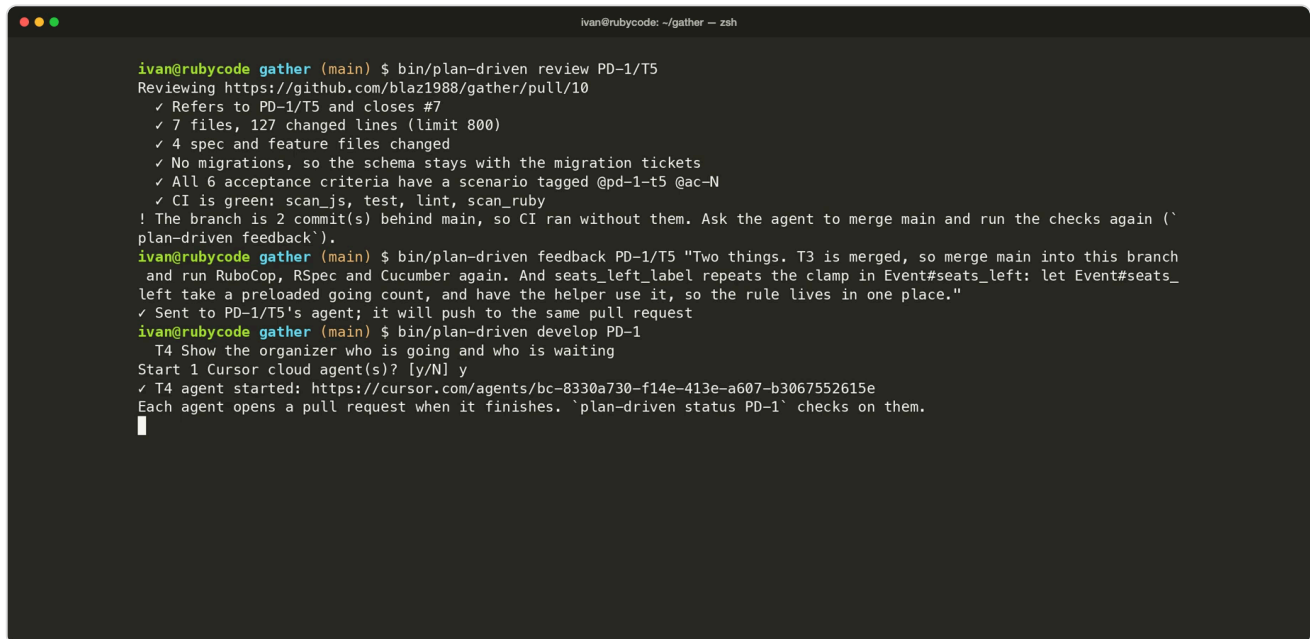
A terminal window with a dark background and light green text. The window title is 'ivan@rubycode: ~/gather — zsh'. The terminal shows a sequence of commands and their outputs. The first command is 'bin/plan-driven review PD-1/T5', which triggers a review of a pull request. The output lists several checks: 'Refers to PD-1/T5 and closes #7', '7 files, 127 changed lines (limit 800)', '4 spec and feature files changed', 'No migrations, so the schema stays with the migration tickets', 'All 6 acceptance criteria have a scenario tagged @pd-1-t5 @ac-N', and 'CI is green: scan_js, test, lint, scan_ruby'. A warning message follows: '! The branch is 2 commit(s) behind main, so CI ran without them. Ask the agent to merge main and run the checks again (`plan-driven feedback`)'. The second command is 'bin/plan-driven feedback PD-1/T5 "Two things. T3 is merged, so merge main into this branch and run RuboCop, RSpec and Cucumber again. And seats_left_label repeats the clamp in Event#seats_left: let Event#seats_left take a preloaded going count, and have the helper use it, so the rule lives in one place."' followed by a confirmation 'y'. The output of the feedback command includes: 'Sent to PD-1/T5's agent; it will push to the same pull request', 'T4 Show the organizer who is going and who is waiting', 'Start 1 Cursor cloud agent(s)? [y/N] y', and 'T4 agent started: https://cursor.com/agents/bc-8330a730-f14e-413e-a607-b3067552615e'. A final note states: 'Each agent opens a pull request when it finishes. `plan-driven status PD-1` checks on them.'

Figure 10. The review notices the branch is behind main. The feedback asks the agent to merge main and to move a duplicated rule onto the model.

Proof, not promises

With all five merged, `plan-driven evidence PD-1` ran the plan's Cucumber scenarios on main. Each of the 33 acceptance criteria had a passing scenario, including the one where several threads race for the last seat and exactly one wins.

```
ivan@rubycode: ~/gather -- zsh

T2.1 passed An event with capacity 3 and no RSVPs has 3 seats left, is not full...
T2.2 passed For an event with capacity 2, four people calling rsvp in turn get ...
T2.3 passed Calling rsvp twice for the same person returns the same record and ...
T2.4 passed When several threads race for the last seat of an event, exactly on...
T2.5 passed Destroying a going RSVP promotes only the oldest waitlisted RSVP to...
T2.6 passed rsvp returns nil and creates no row when the user is the event's or...
T2.7 passed Destroying a going user promotes the next waitlisted person, destro...
T3.1 passed A guest opening "Ruby Zagreb Meetup #42" with 3 seats sees "Seats l...
T3.2 passed A signed-in person who clicks "RSVP" sees "You're going", sees "Sea...
T3.3 passed When 3 people are going, a fourth person sees "Full" and a "Join wa...
T3.4 passed When a going person clicks "Cancel RSVP", they see "Your RSVP was c...
T3.5 passed A waitlisted person who clicks "Leave waitlist" no longer sees a wa...
T3.6 passed The organizer sees no RSVP button on their own event, and nobody se...
T3.7 passed A guest who sends POST or DELETE to /events/:id/rsvp is redirected ...
T4.1 passed Organizer Ana sees "Going (3)" followed by the three attendees' nam...
T4.2 passed Organizer Ana sees "Waitlist (1)" with "#1" next to the name of the...
T4.3 passed After a going attendee cancels, Ana sees the promoted person under ...
T4.4 passed Marko, who is signed in but is not the organizer, sees neither the ...
T4.5 passed A guest sees neither the "Going" nor the "Waitlist" list
T4.6 passed No attendee's email address appears anywhere on the event page, inc...
T4.7 passed The organizer's list shows no controls to remove or reorder people
T5.1 passed An upcoming event with capacity 3 and no RSVPs shows "3 of 3 seats ...
T5.2 passed After one person RSVPs, the events page shows "2 of 3 seats left" f...
T5.3 passed An event where every seat is taken shows "Full" on the events page
T5.4 passed Waitlisted RSVPs do not reduce the seats left shown on the events page
T5.5 passed An event whose capacity was lowered below its going count shows "Fu...
T5.6 passed Rendering the events page with several events runs a single query a...

✓ 33 of 33 acceptance criteria are proven by a passing scenario.
ivan@rubycode gather (main) $
```

Figure 11. Every acceptance criterion, and the scenario that proves it.

plan-driven report PD-1 wrote the delivery report: each ticket with its pull request, merge commit and approver, the proof matrix, the guard findings, every approval and the full timeline. It's the document I'd send to a product owner, and nobody wrote it by hand.

gather/docs/plans/pd-1-rsvps-with-a-waitlist/delivery-report.html

PD-1: RSVPs with a waitlist (delivery report)

Status: delivered · Revision 3 · Created by Ivan Blažević <ivan.blazevic@rubycode.co> · 29 September 2026

Summary

5 of 5 tickets merged, 33 of 33 acceptance criteria are proven by a passing scenario.

Tickets and pull requests

#	Ticket	Status	Pull request	Merge commit	Approved by
T1	Migration: Create rsvps table	merged	#8	ba1bdf1	Ivan Blažević <ivan.blazevic@rubycode.co>
T2	Add Rsvp model and Event rules for RSVP, cancel, waitlist and promotion	merged	#9	a30c1ab	Ivan Blažević <ivan.blazevic@rubycode.co>
T3	Let signed-in people RSVP, join the waitlist and cancel from the event page	merged	#11	75bf79d	Ivan Blažević <ivan.blazevic@rubycode.co>
T4	Show the organizer who is going and who is waiting	merged	#12	1bdb205	Ivan Blažević <ivan.blazevic@rubycode.co>
T5	Show seats left on the events index	merged	#10	09b73f3	Ivan Blažević <ivan.blazevic@rubycode.co>

Acceptance criteria and proof

Cucumber, run 29 Sep 2026 22:43 on commit 1bdb205: bundle exec cucumber --tags "@pd-1-t1 or @pd-1-t2 or @pd-1-t3 or @pd-1-t4 or @pd-1-t5"

Figure 12. The delivery report, generated from the audit trail.

The feature

And the feature works the way the plan says. Three people take the three seats, the fourth joins the waitlist as number one, and when someone cancels she's promoted automatically. The organizer sees who's going, in order, with names only.

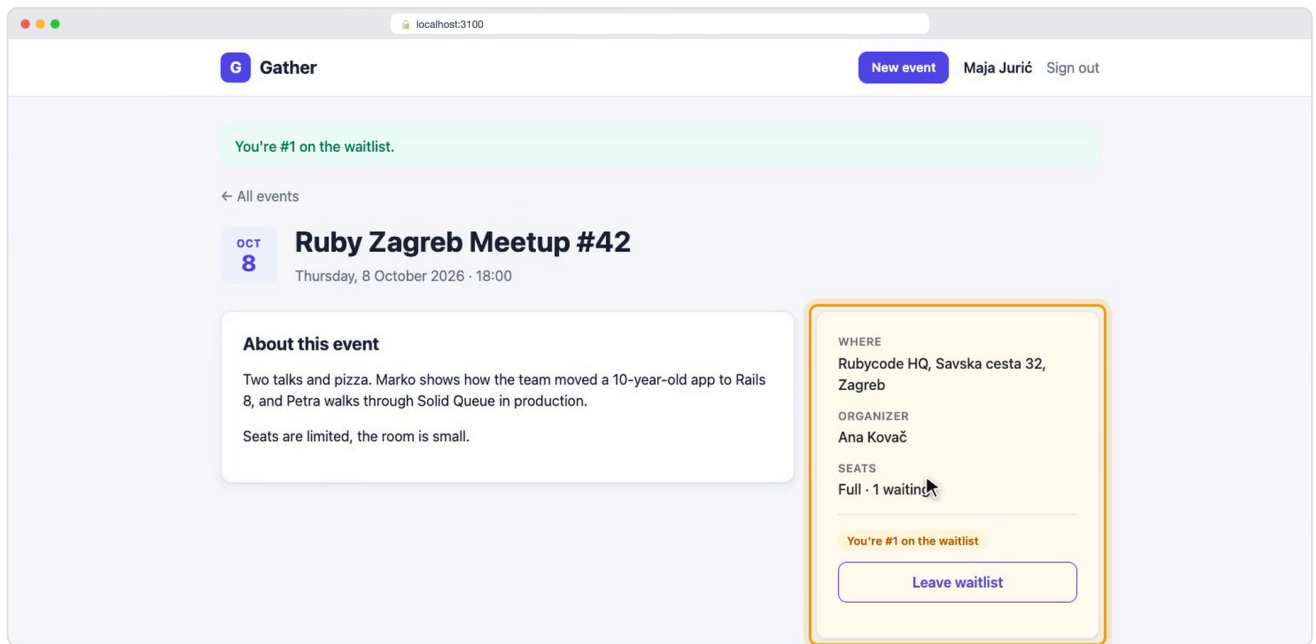


Figure 13. The event is full, so Maja joins the waitlist.

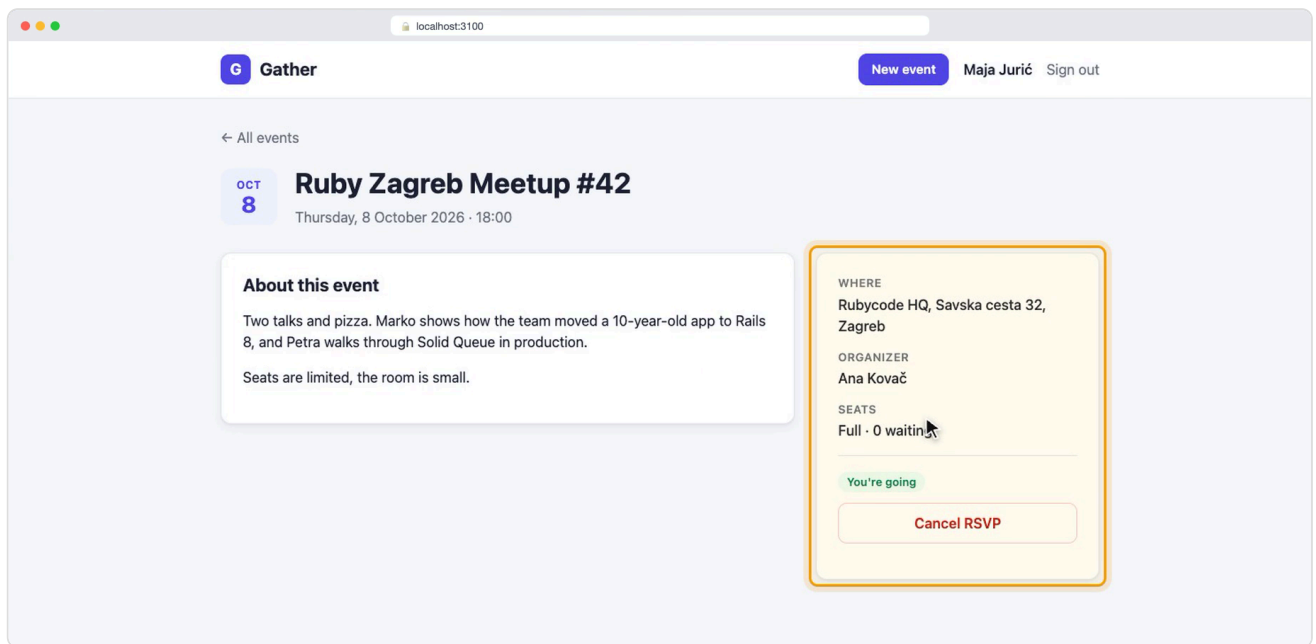


Figure 14. Petra cancels, and the seat goes to the first person waiting.

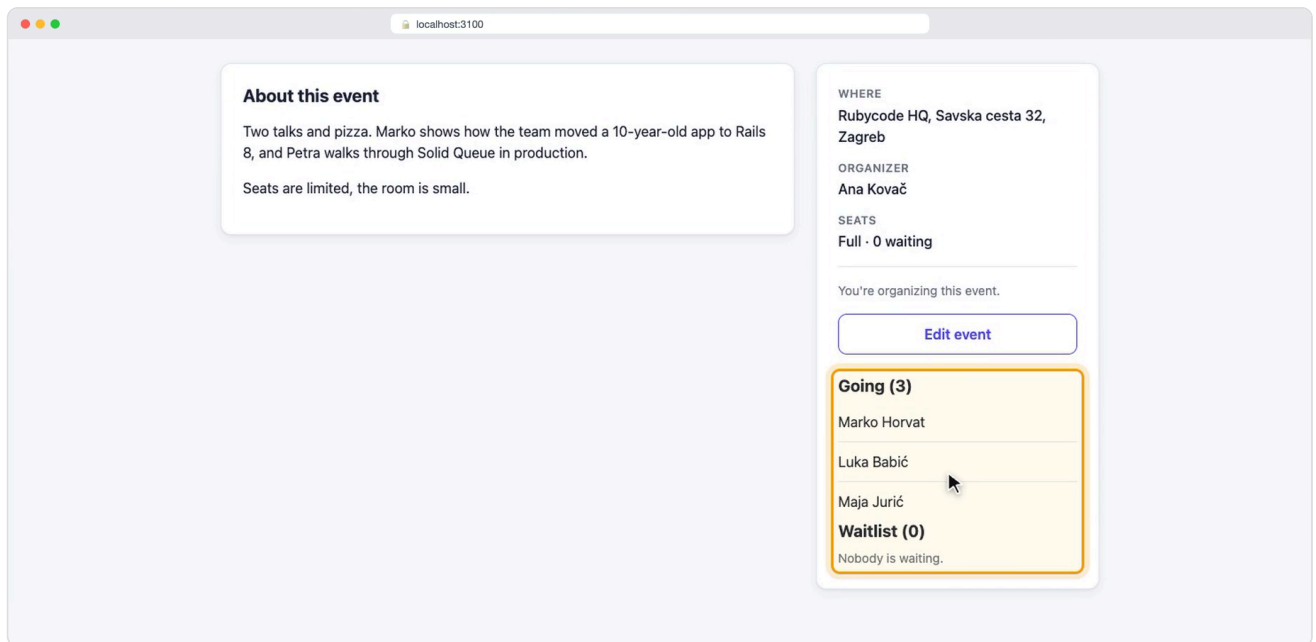


Figure 15. The organizer's view: who's going, in the order they RSVPed, and who's waiting.

On main, Gather now has 71 specs and 49 Cucumber scenarios, all green, and RuboCop is clean.

Where the time went

Once the tickets are approved, every step lands in the audit trail with a timestamp: an agent starting, its pull request opening, feedback, approval, the merge. So the gem can show where the time actually went, without anyone filling in a timesheet and without asking a model.

Every plan in the wizard has a Statistics page, `plan-driven stats` prints the same numbers in the terminal, and the delivery report has a Statistics section with the charts. This one is from a second feature in the same app, comments on events: six tickets, 42 of 42 acceptance criteria proven, and 1 h 20 min from approving the tickets to delivery.

[← PD-3 Comments on events](#)

Statistics

delivered Computed from the audit trail: every number is a recorded event, nothing is estimated.

IDEA TO DELIVERY
1 h 25 min
plan drafted to delivered

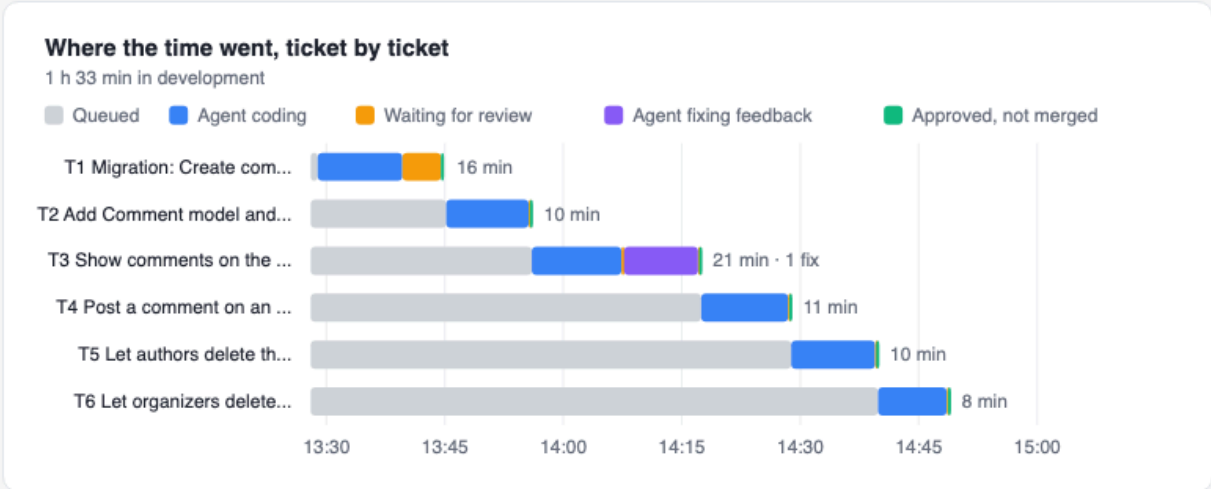
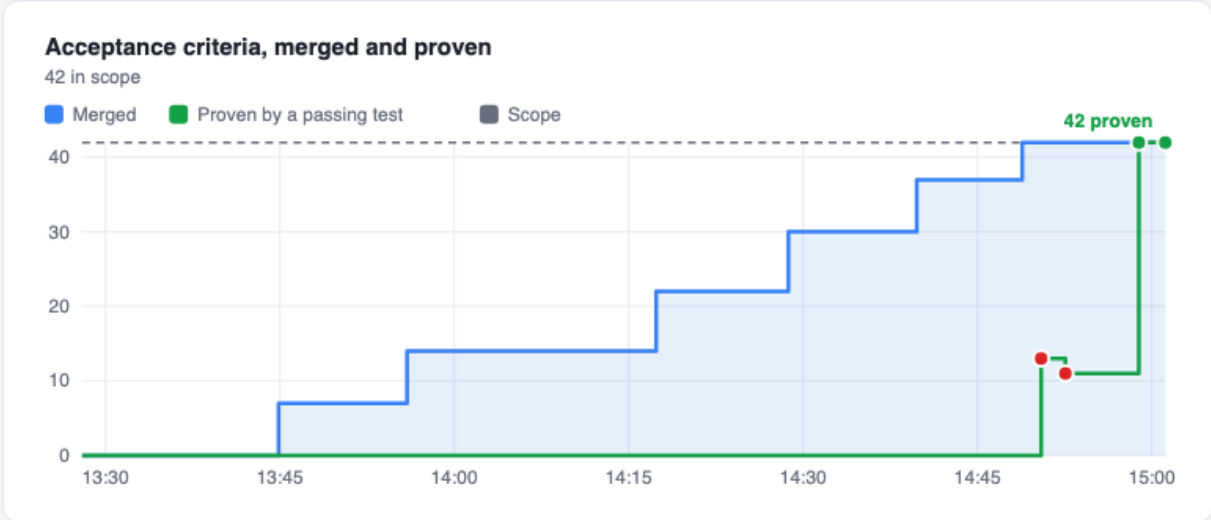
DEVELOPMENT
1 h 20 min
6 of 6 tickets merged

CRITERIA PROVEN
42 / 42
by a passing scenario

APPROVED THE FIRST TIME
5 / 6
1 review round

AGENTS' SHARE OF THE WORK
91%
the rest is waiting on people

TOKENS
7.62M
set config.token_prices for dollars



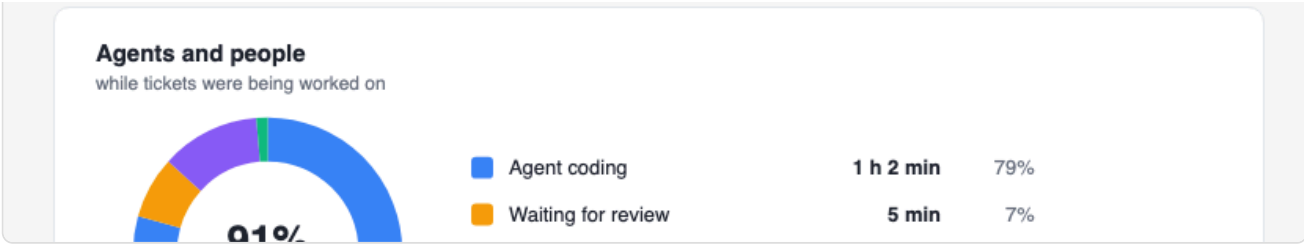


Figure 16. The Statistics page: acceptance criteria merged and proven over time, and each ticket's time on a shared clock.

Two things stand out. The green line, criteria proven, only rises at the end: the first two evidence runs failed, and the third proved all 42. And agents took 91% of the time the tickets were being worked on, while pull requests waiting for review took 7%. On a plan where that donut is mostly amber, the time is going into review, not into writing code.

The delivery report also marks every acceptance criterion's result in colour, so a product owner sees at a glance what's proven and what isn't.

T6	2	1 h 11 min	8 min	8s	-	10s	0	8 min
----	---	------------	-------	----	---	-----	---	-------

Acceptance criteria and proof

Cucumber, run 30 Sep 2026 15:01 on commit efe49c9: bundle exec cucumber --tags "@pd-3-t1 or @pd-3-t2 or @pd-3-t3 or @pd-3-t4 or @pd-3-t5 or @pd-3-t6"

AC	Criterion	Scenario	Result
T1.1	Migrating creates the comments table with NOT NULL columns event_id, user_id, body, created_at and updated_at, and body has no default	Migrating creates the comments table (features/pd-3-comments-on-events/t1.feature:11)	✓ passed
T1.2	Inserting a comments row for an event or user that does not exist fails with ActiveRecord::InvalidForeignKey	Rows must point at an existing event and person (features/pd-3-comments-on-events/t1.feature:17)	✓ passed
T1.3	Inserting a comments row with an empty or space-only body fails the check constraint comments_body_length_check	The body must not be empty or only spaces (features/pd-3-comments-on-events/t1.feature:24)	✓ passed
T1.4	Inserting a comments row with a 1,001-character body fails the check constraint comments_body_length_check, and a 1,000-character body containing accented letters and emoji is accepted	The body is at most 1,000 characters (features/pd-3-comments-on-events/t1.feature:31)	✓ passed
T1.5	The index index_comments_on_event_id_and_created_at is on event_id, created_at, the index index_comments_on_user_id is on user_id, and there is no index on event_id alone	Event page and per-person lookups are indexed (features/pd-3-comments-on-events/t1.feature:39)	✓ passed
T1.6	Neither comments foreign key cascades on delete: deleting an event or user that has a comments row directly in the database fails with ActiveRecord::InvalidForeignKey	Deleting an event or person does not cascade to comments (features/pd-3-comments-on-events/t1.feature:45)	✓ passed
T1.7	Rolling back the comments migration drops the comments table and leaves the events, users, rsvps and sessions tables unchanged	Rolling back drops only the comments table (features/pd-3-comments-on-events/t1.feature:55)	✓ passed

Figure 17. Every criterion with the scenario that proves it, in the delivery report.

Statistics are new in 0.3.0.

The rules live in code, not in the prompt

The easiest way to build a tool like this is a long prompt: "always write tests, keep pull requests small, never drop a column in one step". Models follow instructions like these most of the time. "Most of the time" is fine for a draft and not fine for a merge.

So in `plan_driven`, the model writes and Ruby checks. Guards are plain Ruby classes that run at every step:

- **PlanGuard** checks that every required section is written, that security ends with a risk level, and that Existing Data Structure cites only models, files and columns that really exist.
- **MigrationGuard** refuses removing or renaming a column in one step unless the plan stages it, and warns about NOT NULL without a default and non-concurrent indexes on PostgreSQL.
- **TicketNormalizer** corrects anything with one right answer, like title prefixes, estimates rounded to your scale and dependencies renumbered, and reports each correction.
- **TicketGuard** checks acceptance criteria, estimates, dependency cycles, the order of expand and contract on each table, and that every table the plan changes is covered by a ticket.
- **PrGuard** checks the pull request's scope and size, that it has specs, that only migration tickets add migrations, that every acceptance criterion has a tagged scenario, that CI is green and that the branch isn't behind main.

When a guard fails on a draft, its errors go back to the model as a list to fix. A plan that still fails isn't accepted. Nothing merges while a guard fails or CI is still running.

Guards don't replace reading the code. They make sure that when you read a pull request, you're reading for design and behaviour, not checking whether someone remembered the tests.

What it asks of you

`plan_driven` gives your attention to decisions, and takes it away from bookkeeping. In this run, my part was answering the interview, reading the plan, deciding the open questions, steering the tickets, reading five pull requests, trying the UI and sending feedback twice. The agents wrote the code, and the gem kept the paperwork.

It also has requirements, and it's better to know them upfront:

- The app needs to be on GitHub with CI on pull requests, and the agents are Cursor cloud agents, so you need a Cursor account with GitHub connected.
- Evidence needs Cucumber. The first agent can set it up if your app doesn't have it.
- The plan's state lives in the database of whoever drives it. The documents go to `docs/plans/`, which you commit, so the whole team reads the same plan and report.
- For drafting, use a strong model. Planning is where it pays for itself. We used Claude Opus 5.5 through Cursor, and OpenAI and Anthropic work too.

Try it

```
# Gemfile
gem "plan_driven", group: :development
```

```
bundle install
bin/rails generate plan_driven:install
bin/rails db:migrate
bundle exec plan-driven configure      # keys go to ~/.plan_driven/config, never into the app
bundle exec plan-driven doctor
bundle exec plan-driven new "Your next feature"
```

The [README](#) walks through setup and the whole run step by step, with the real output. The Gather repository has the [plan](#), the [tickets](#) and the [delivery report](#), and [pull requests #8 to #12](#).

- Code: github.com/blaz1988/plan-driven (MIT)
- Gem: rubygems.org/gems/plan_driven
- Ruby 3.1 to 3.4, Rails 7.0 to 8.1

It's the first release. If a guard your team relies on is missing, or the drafter got your plan wrong, [open an issue](#). We'd like to hear about it.

About Rubycode

`plan_driven` is built and maintained by [Rubycode](#), a Ruby on Rails company from Zagreb. We build and rescue Rails applications, and help teams put AI agents to work safely. **Need Ruby or Ruby on Rails engineers? Get in touch.**

- Ivan Blažević, creator of the gem: ivan.blazevic@rubycode.co · [+385 99 351 3642](tel:+385993513642) · [LinkedIn](#)
- Rubycode: info@rubycode.co · [+385 98 980 1880](tel:+385989801880) · rubycode.co