Autumn SCM: Fir, Aspen and Willow

Lars Cowé, Abdellah El Morabit, Lucas Leonte and Luke Bouvard

Autumn SCM is a suite of three microservices (Fir, Aspen and Willow) that reimplements source control management (SCM) for agent-native development. It stays fully compatible with existing SCM systems.

MVP

Three parts that work together or on their own:

- Fir, our local agent for workspaces and parallel agents
- Aspen, agent-native SCM
- Willow, our translation layer to existing SCM systems (Colruyt)

The MVP we deliver in March, on one GitLab project with Java code:

- **Fir**a copy-on-write workspace per agent with its own test database on one server; logs per workspace, reachable only through GitLab job logs; redaction of secrets and personal data and static analysis after every fix; tests, checks, time, cost and memory reported on every merge request.
- **Aspen**receives the patches from Fir and resolves conflicts; cost and KPIs per merge request, project and week on our own dashboard, with an overview and drill-down.
- **Willow**the GitLab adapter; labels, priority and team for new issues from fixed rules; a merge request with a fix for small, well-scoped issues; comments on changed lines; a new commit for one review comment; one risk label per merge request from fixed rules; a person writes the acceptance criteria on the work item; one local and one API model behind the same interface, with a fixed model per task.

Description

Colruyt's brief

Colruyt asks us to compare GitHub Copilot with GitLab Duo, and to then build similar functionality on its **free, self-hosted GitLab**, without paid Copilot or Duo features. The brief names **four use cases** as a starting point: merge request review, fixing review comments, issue triage and issue fixing. The solution should use open-source parts and **AI models that can be replaced**, stay independent of the platform where practical, and be useful, secure and maintainable. The selected use cases are then built, connected to Colruyt's development workflow, and evaluated. Besides that, they care about the cost because of the scale things run at within their company.

Our ideas

On top of the brief, we tackle a second problem: when several coding agents work in the same codebase at the same time, they get in each other's way, and their changes come faster than people can review them. Our answer starts with **a workspace per agent** and adds four ideas on top of it.

Agents make this a problem for every team, not only for the largest companies. At Anthropic, merges per day went from about 2,500 to 30,000 between May and September 2026, and they rebuilt both their source control and their review for agents. The tools that exist today either **isolate too little** (like a git worktree that still shares ports and databases) or **copy too much** (like a full container per agent).

The workspace per agent sits in between: it branches code, data and services from one shared base and **copies only what an agent changes**. The other four ideas build on this. With **intent up front**, a person confirms the acceptance criteria before the agent starts. **Guardrails** keep secrets and personal data away from the model and check every fix.

Colruyt's feedback

We presented our scope to Colruyt on 2 November 2026. The word that came back most was '**agnostic**'. The solution must work with any AI model and any platform, which rules out a vendor SDK, such as the Claude SDK, or a custom-trained model. Moving from the GitLab API to the GitHub API may require changing a few lines, but never the core logic.

Cost matters most to Colruyt, in line with its low-price strategy. Every run is measured down to the tokens; the benefit must be clearly higher than the cost, and that is also why the platform they use is GitLab: it is free. **AI is only used where it adds something**, because plain automation is often enough, and a decision such as the risk score comes from fixed rules.

The setup can stay modest. **A few agents in parallel are enough**, there is no need to design fifty, and since most of the code is Java, that is where we start. Triage brings the most value to GitLab issues. **Security**, on the other hand, **has to be built in**, or at least prepared in a framework, so the result can work in a corporate environment.

Every idea can also have **a light and a heavy variant**. For the collected data, the light variant is our own dashboard, and the heavy one makes the data available to tools such as Grafana or Kibana. Either way, the numbers **have to be visible** beyond a single merge request.

Colruyt's use of GitLab is still at an early stage. **Merge requests are already common, but issues and pipelines that review merge requests are not**, so the project should also make those worth using.

Colruyt is involved throughout. They gave us the brief and gave feedback on our scope on 2 November 2026. We build and run the services on their GitLab, connected to their development workflow, and we set the quality and cost targets together with them. The result is evaluated against those targets.

Scope

We build **three modular services**: **Fir**, **Aspen** and **Willow**. Together they run the four use cases on Colruyt's GitLab, give every agent its own workspace, and show the cost and what ran for every change.

Fir is the local agent that handles the workspaces and the parallelism between agents. It contains a **read-only buffer** that cannot be modified locally. Instead, Fir generates patches and sends them to Aspen. It is built in Rust.

Aspen handles the incoming patches, and conflicts are resolved there. **Pulling from Aspen is the only way to resolve a local conflict.** Aspen is built in Rust.

Willow is the SCM connector. It handles the SCM APIs and carries the four use cases from Colruyt's brief on GitLab: merge request review, fixing review comments, issue triage and issue fixing. Triage can be done with a decision model. Willow is built in Java.

For each of the four use cases, the analysis decides how to solve it: with fixed rules, with a single model call, or with an agent that runs code. Wherever an agent needs to run code and tests, the workspace gives it a safe place to do so.

What is in scope

- **Copilot and Duo comparison**: Our own services measured against the documented Copilot and Duo features on the same use cases
- **Services and adapters**: Extra adapters, showing the core logic stays the same
- **Model interface**: Several models (different types, for example vector models) compared on quality and cost.
- **Model per task**: The services pick the model for each task from measured quality and cost, with automatic model routing
- **Issue triage**: AI suggestions where the rules cannot decide, duplicate detection, learning from corrections, and a first attempt to reproduce a bug with a unit test
- **Issue fixing**: Larger issues that touch several modules
- **Merge request review**: A review that also reads the evidence and acceptance criteria, and drops comments nobody finds useful
- **Fixing review comments**: Every round of comments on the same merge request
- **Workspace per agent**: The whole environment, with services, data snapshots and shared test results
- **Workspace logs**: A secured log view per workspace, with redaction and a retention period
- **Guardrails**: Dependency and licence audit, and protection against hidden instructions in issues
- **Evidence**: A full trace of every step the agent took, which can be replayed
- **Risk score**: Rules tuned on Colruyt's own merge request history
- **Intent up front**: AI drafts the criteria, a person confirms them, and they become tests before the agent starts
- **Metrics and cost**: Data exported to Grafana or Kibana for a view across all teams and projects, with an alert when cost or quality drifts
- **Overview and drill-down**: Search and filters across all tasks, by project, issue or merge request

Possible extra ideas:

1. Cost Control:

Every model and tool call is traced with OpenTelemetry. Before a task starts, its tokens and cost are predicted from similar earlier tasks. A task stops when it passes its token or time budget, and when usage runs high the services switch to a cheaper model or sends less context, known as model routing.

2. Review budget per risk

The risk score also decides how much a review may cost. Changes to only documentation or lockfiles are skipped, small changes get one pass with a cheap model, and only risky changes go to the strongest model.

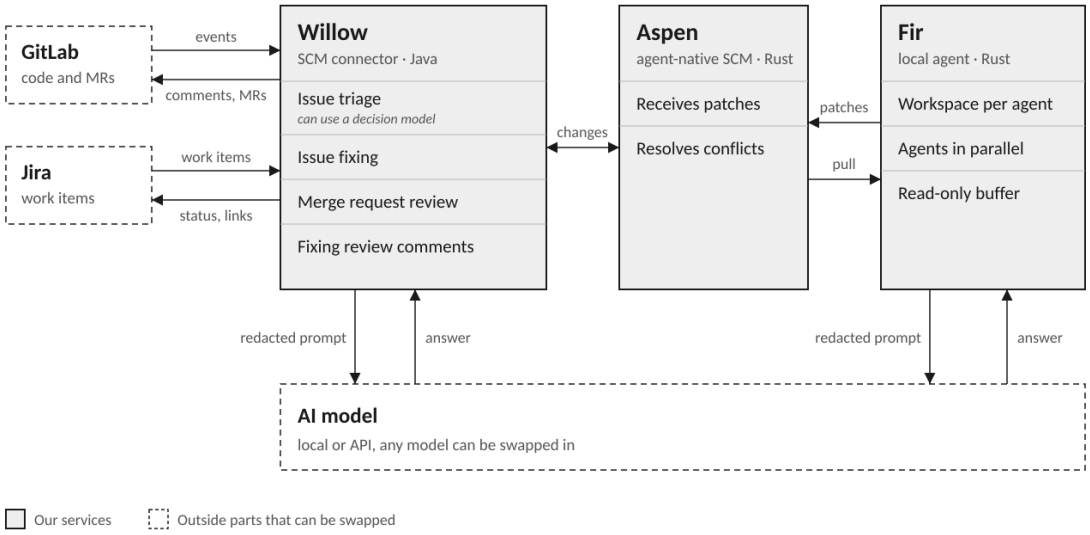
3. Reproduce first

Before the agent fixes a triaged issue, it reproduces the problem with a failing unit test. A person approves that test before the agent spends time and tokens on a fix, so no budget goes to half a solution or none at all.

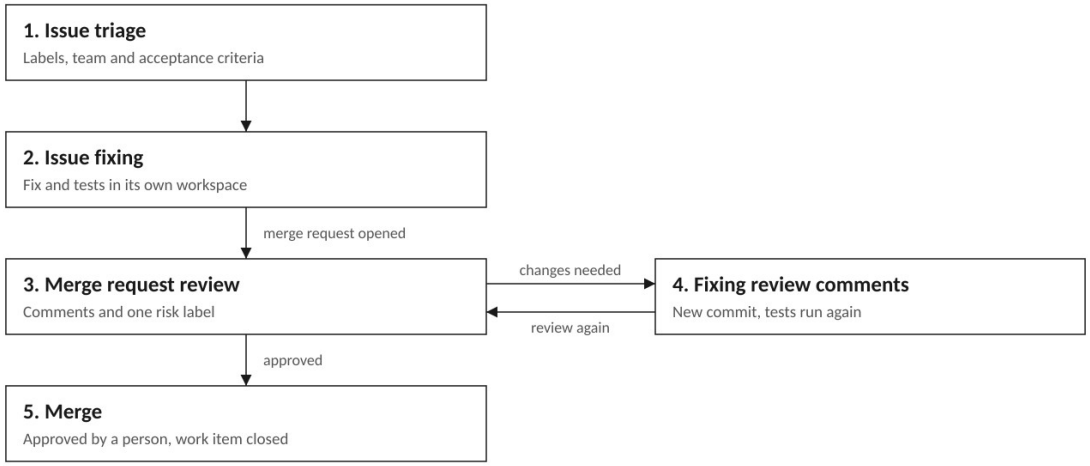
Out of scope

1. Replacing git or GitLab
2. Training a custom model
3. Tying the services to one AI provider's SDK
4. Designing for dozens of agents on one codebase at once
5. Merging without a person

How it fits together



Every part is a separate module with its own interface, in a **ports-and-adapters architecture**, also known as hexagonal architecture. The core logic **never talks to GitLab, Jira or a model directly**, only through an adapter, so each of them can be swapped without changing the use cases. Modules can be replaced, turned off or deployed on their own, and the workspaces, which need the most resources, can run as a separate service on their own server.



The four use cases follow each other in one flow, and a merge request can go round the review loop several times.

How we measure it

We measure the result with the **six KPIs** from the scope presentation: fix rate, review quality, triage accuracy, workspace start time and parallel agents, guardrail leaks and risk score safety, with the **cost per merged fix** next to them. The targets are set together with Colruyt.