YC’s QM Agent Harness in Practice: What the MIT License Buys You and What It Does Not

Y Combinator’s accounting and legal functions run on an agent harness that shipped at version 0.1.4, and on July 31, 2026 they handed you the source. The repo has a security mode called “Dangerous.”

THE SHORT VERSION

Y Combinator open-sourced QM on July 31, 2026 under MIT, a multiplayer agent harness it runs internally across accounting, legal, events and engineering. The repo hit roughly 1,900 GitHub stars within hours, at v0.1.4. MIT buys you the code and the architecture. It buys you no warranty, no SLA, and no answer when a scoped workspace holding a keychain view does something you did not expect.

QM ships the scaffolding around an agent, not the agent

YC describes QM as a "multiplayer agent harness for work. In Slack and on the web." The word that matters is harness. QM ships identity, state, permissions, scheduling and the surfaces people actually work in.

The architecture, as reported in the MarkTechPost write-up and a source-code read, is a headless core that multiple agent backends can drive: Pi, OpenCode, Codex and Claude Code. State persists in Postgres. On top of the core sit a Slack app and a web UI, with optional admin and portal plugins. Each employee and each Slack room gets its own scoped workspace carrying memory, files, a keychain view, permissions, crons and durable sandboxes.

That last sentence is the whole product. Most agent frameworks give one agent one context. QM gives an organization many contexts, each with a boundary, and makes the boundary the unit of administration. The scoping is per person and per room, so a conversation in #legal and a conversation in #eng do not share memory or credentials by default.

QM is MIT-licensed and self-hostable at github.com/yc-software/qm. One catalog, EveryDev, records public release on July 29, 2026 and v0.1.4 by July 31. That is not 1.0, and YC's own framing says so: "an experiment," and "it's early and has bugs."

QM assumes forty people in Slack, not one operator at a terminal

Nearly every agent framework released in the last two years assumes a single operator at a terminal or a single service behind an API. QM assumes forty people in Slack who each need an agent that remembers their own work, holds their own credentials, and cannot reach into someone else's.

That is a harder problem than it sounds, and it is why the feature list reads like infrastructure. Crons mean scheduled work without a human in the loop. Durable sandboxes mean execution that survives restarts. A keychain view makes credentials a

scoped object rather than environment variables in a config file. Postgres means you can inspect and back up the state, which matters more than it should in a category where plenty of tools keep agent memory somewhere nobody can audit.

The multi-backend design is the most defensible engineering choice here. Pi, OpenCode, Codex and Claude Code are four different bets on how agents should work, and QM treats them as interchangeable drivers behind a headless core. If your backend of choice degrades, gets repriced, or changes its tool-calling semantics, you swap the driver instead of the platform. Most frameworks do not offer that hedge.

The proof point YC is selling is dogfooding: four internal functions run on QM, including the engineering of QM itself. Accounting, legal, events, engineering. I weight that reasonably highly, because a framework whose authors depend on it for their own accounting workflows has a different bug-fix incentive than a demo repo. But be precise about what dogfooding proves. It proves QM works for YC's org shape, YC's Slack conventions, and YC's tolerance for a tool its own engineers can patch same-day. It does not prove it works for yours.

What "Dangerous" mode actually means for your threat model

QM exposes three security modes: Strict, Auto and Dangerous. The names are honest, which I appreciate, and they are the clearest signal in the whole repo about who this is for.

Naming a mode "Dangerous" shifts liability to the operator by making the risk legible. Better that than permissive defaults hiding behind a flag called `–fast`. It is still your problem once you flip it. The threat-model analysis from BitsMinds puts the deployment point plainly: QM is org software rather than a desktop app, and it expects infrastructure competence to deploy.

Stack the primitives and the exposure is easy to see. Scoped workspaces hold a keychain view. Crons fire without a human present. Durable sandboxes execute code. Slack is the input surface, so anything that lands in a channel is a potential instruction path to an agent that holds credentials and can run jobs on a schedule. This is the standard agent injection surface, and I have argued before that tool-call filtering is the layer where you actually get

to intervene. QM gives you permission scoping, which is necessary. Nothing in the reported feature set suggests it gives you content-level inspection of what flows into a tool call, and that is a different control.

One more thing to check before you wire anything external in. The MCP specification finalized on 2026-07-28, three days before QM's announcement, and it makes MCP servers formal OAuth 2.1 resource servers while removing sessions. Any harness plumbing MCP tools inherits that change. I have not seen reporting on how QM's MCP wiring, if any, lines up with the finalized spec, so given the release timing I would treat compliance as unverified rather than assumed. If you connect QM to MCP servers, the auth posture of those servers is your audit, not YC's. That matters because the overwhelming majority of MCP servers in the wild ship without auth at all.

Roughly 1,900 stars in hours, then two quiet months

The strengths are short to list. The multi-backend core removes single-vendor dependency at the agent layer. Postgres-backed state is inspectable, backupable and familiar to any ops team. Workspace scoping is per person and per room, with explicit permissions and a keychain view. YC runs it internally across four functions, including its own engineering. And MIT means you can fork it, strip it and ship it with nobody to negotiate with.

The rough edges are the other half of that. It was v0.1.4 as of July 31, 2026, and YC's own words are "it's early and has bugs." No major feature release or version jump was reported through August or September 2026 beyond documentation updates. A "Dangerous" security mode exists and the operator owns everything that happens in it. Deployment assumes real infrastructure competence. MIT also means zero warranty, zero support SLA and no roadmap commitment to you.

The quiet entry there is the version silence. Roughly 1,900 stars within hours, per the coverage, then two months with no reported version jump. Stars measure attention on day one. They say nothing about whether a maintainer will look at your issue in November.

My read on the silence: it is ambiguous, and I would not over-interpret it. A 0.1.x project used internally by its authors may simply be getting patched in ways that generate no

coverage, and absence of reported releases is not absence of releases. If you are deciding whether to build on this, check the commit log and the issue response times instead of the star count.

Adopt QM for the architecture, not for the maintenance contract

Run QM if you have a platform or infra team that already operates Postgres, already deploys internal services, and is willing to own a fork. The reason to adopt at 0.1.4 is that you want the architecture more than you want support. Read the source, take the workspace scoping model, take the headless multi-backend core, and accept that when something breaks at 2am it is yours.

Skip QM if you need a supported product. MIT gives you zero cost and zero recourse in the same clause. If your compliance posture requires a vendor to answer for an incident, an experimental 0.1.x repo from a startup accelerator is not that vendor, and no number of GitHub stars changes it.

WHERE I LAND

My take: the most valuable thing YC shipped here is a published opinion about what org-level agent infrastructure should look like, with scoped workspaces as the unit, credentials as a scoped object, and agent backends as swappable drivers. I would read the source before I would deploy it. If you deploy it, start in Strict, keep it off anything with money or contracts in it, and treat every Slack message as untrusted input, because that is what it is.

A middle path, if you want one: fork it, run it internally on a low-stakes function first (events scheduling rather than accounting), and instrument every tool call before you widen the scope. QM's own deployment story at YC covers four functions, and I would bet accounting was not the first.

WRITTEN BY

Artur Markus

Software engineer in Basel building AI infrastructure and agentic systems, with a background in pharmaceutical automation, GxP compliance, and validation. I help teams turn AI from a chatbot into a reliable operational layer.

Book a meeting →