

Instructions



 **UPDATE — 2026-09-03: We've rebranded Grader Guidance to Holistic Rubric. Going forward, we'll collect an atomic rubric alongside it as well.**

We've rebranded Grader Guidance to **Holistic Rubric**. The document itself hasn't changed. It is still the task-specific grading notes you write for the grader, and every rule about what goes in it still applies. On new tasks it lives at `tests/holistic-rubric.md` , and you draft it with the `/write-holistic-rubric` skill. Going forward, we'll also collect an **atomic rubric** alongside it. The atomic rubric breaks your holistic rubric down into a list of small criteria that the grader judges one at a time. Once your holistic rubric is final, the new `/write-atomic-rubric` skill generates it for you, and both rubrics ship with every finalized task. The **Writing the Holistic Rubric** and **Generating the Atomic Rubric** sections walk through both documents.

New tasks are now graded by Claude Fable 5.1. A task you already have in progress keeps the grader it started with, and you can finish it as is. If you'd rather move it onto the new grader, copy `task-shared/test.sh` into your task's `tests/` folder and regrade your reference runs. Everything else in this release is listed in  **Recent Changes**.

Please download the new toolkit before you start your next task.

Raccoon — Code Execution Task Creation

 You are being given access to models via a proxy for this project. You are not permitted to use these models for any purpose that does not support completion of a task for this specific project, and all traffic through the proxy is logged and monitored. Abusing model access will be subject to penalties, including but not limited to removal from the platform.

 **New to the project?** Start with a [2-minute overview of what we're doing](#), put together by J.D Nichols.”

Quick Navigation:

[Confidentiality](#) | [Start Here](#) | [Toolkit](#) | [Repositories](#) | [Agents](#) | [Meaningful Failure](#) | [Snapshots](#) | [Workspace](#) | [instruction.md](#) | [Grading](#) | [Holistic Rubric](#) | [Detectors](#) | [Reference Runs](#) | [Atomic Rubric](#) | [Regrade](#) | [Submitting & Feedback](#) | [Versions & Migration](#) | [Examples](#) | [Troubleshooting](#)

Quick Links:

- [Submission Overview](#): View your current standing, reported-hours progress, review-queue outlook, and submission history.
- [Grading Standard](#)
- [Workflows](#)
- [FAQ](#)
- [Searching for model failures: hardness ladder](#)
- [~20min video of Nick talking about tasks that he likes](#)
- [Learning Exercise refresher](#)
- Task Catalog ([JSON](#)) ([Viewer](#))

▼ **Recent Changes**

Dated project updates, newest first. Each entry points you to the section with the full details.

- **2026-09-05 — Rubrics must grade any arbitrary run; unfired meaningful failure modes stay; detectors are advice.** Write the holistic rubric around what makes a good response and what makes a bad response on this task, in general terms, so it can grade any run, not just yours. Don't remove a meaningful failure mode because your runs don't show it; only the failure the task targets has to fire. For detectors: re-run them when their inputs change (the **Detectors** section now lists exactly when), once, after your last edit. If you don't agree with a finding, don't keep re-running the detector hoping for a clean or high-confidence verdict — tick the false-positive box on the Test page, say why, and submit. See **Writing the Holistic Rubric**, **What Makes a Failure Meaningful**, and **Detectors**.
- **2026-09-03 — We've rebranded Grader Guidance to Holistic Rubric, and every finalized task now ships an atomic rubric as well.** The holistic rubric is the same document with the same rules. On new tasks it lives at `tests/holistic-rubric.md` , and the `/write-holistic-rubric` skill replaces `/write-grader-guidance-consolidated` . Once your holistic rubric is final, the new `/write-atomic-rubric` skill generates `tests/atomic-rubric.yaml` and `tests/grader-context.md` , and both rubrics ship with a finalized submission. Two new detectors check the atomic rubric, which brings the set to seventeen. New tasks are graded by Claude Fable 5.1. A task you already have in progress keeps its grader; to move it onto Fable 5.1, copy `task-shared/test.sh` into your task's `tests/` folder and regrade your reference runs. Tasks created on earlier toolkit versions keep their filenames and their grader, so never rename a committed task file. Packaging now stops when you have copied four or more runs and fewer than four of them are clean. If you have not yet submitted your task for feedback, migrate it to this release. Tasks already submitted, or iterating on reviewer feedback, can finish on the version they started with, and a task in that position can be finalized without an atomic rubric. See **Writing the Holistic Rubric**, **Generating the Atomic Rubric**, and **Toolkit Versions & Migration**.



- **2026-09-03 — Codex is now the default agent for new tasks.** Start new tasks in Codex, which runs `gpt-5.6-sol` at maximum reasoning effort. Claude Code remains available if you prefer it, and grading is unchanged either way. See **Choosing Your Agent**.
- **2026-09-01 — Your personal [Submission Overview](#) is now live.** It brings your standing, reported-hours progress, review-queue outlook, and submission history together in one place. Open it from Quick Links whenever you want to check your current information.
- **2026-08-31 — Task-specific tests are expected, and `tests/test-commands.sh` is where they get wired in.** If the repository's suite does not cover the behavior your task turns on, add grader-only specs under `tests/` and register them in `tests/test-commands.sh`. Your task's copy of that file is yours to edit. See **Writing the Holistic Rubric**.
- **2026-08-27 — These instructions are the source of truth.** When we announce a change in Slack, it becomes official once it lands on this page, and every substantive change gets a dated entry in this section. If a Slack post seems to contradict this page, please flag it in the questions channel.
- **2026-08-27 — A heavy penalty can target a criterion, the overall score, or both.** Earlier wording suggested you had to choose one or the other. You can direct a penalty at a named criterion, at the overall score, or at both at once. See **Writing the Holistic Rubric**.
- **2026-08-27 — Early feedback is deprioritized while the review queue is long.** Submit tasks as complete once you have four reference runs and the failure shows in at least one in four of them. Work-in-progress submissions still go through, but they wait behind finalized ones and can take weeks to get a response. See **Submitting & the Feedback Loop**.
- **2026-08-25 — The failure must show in at least a quarter of your reference runs.** With four runs, that is at least one. A task where no run shows the failure is not accepted. See **What Makes a Failure Meaningful**.
- **2026-08-24 — Reviews are bounded.** Each review has a time limit, and a task gets at most four full reviews. Feedback on a work-in-progress version does not count toward the four. See **Submitting & the Feedback Loop**.
- **2026-08-24 — Feedback arrives on the platform.** Reviews come back on the same task instead of as replies in a Slack thread. If a review finds something, the task returns to you with the feedback on it. The **Submitting & the Feedback Loop** section describes the cycle.

▼ Older updates

These entries predate the changes above and are already reflected throughout the instructions.

- **2026-08-10 — Heavy penalties are written in words, not numbers.** Instead of "deduct 0.4", you write "apply a heavy penalty", and the grader decides how much to subtract. See **Writing the Holistic Rubric**.
- **2026-08-07 — The Grading Standard and Codex arrived.** Grading moved to the eight-criterion Grading Standard, with correctness evaluated inside the criteria, and Codex became an authoring agent alongside Claude Code.
- **2026-08-04 — You no longer need to submit early to report time.** You can report time on any in-progress task in the platform's Report time section. The end-task report page is only for unresolved blocking issues.
- **2026-07-27 — Confidentiality.** Everything we provide is confidential and stays on your local machine. The **Confidentiality** section states the policy.
- **2026-07-05 — Heavy penalties replaced score caps.** Dealbreakers are written as heavy penalties rather than caps on the score.

▼ Confidentiality

Everything we provide on this project is strictly confidential and is for your use inside the project only. **All task artifacts should exist solely on your local machine.** This covers, among other things:

- The toolkits and devcontainers provided to you
- The repositories themselves, plus any snapshots, branches, or diffs of them
- The Task Catalog, the Grading Standard, the Workflows, and any other linked documents
- Prompts, holistic and atomic rubrics, admin feedback, example submissions, and material shared in office hours
- The signup link in the logbook. It is not for sharing; referrals for people already on the platform go to an Admin by direct message.
- Office-hours sessions. We record them and link the recordings, notes, and transcripts in the logbook, so do not bring AI note-takers or recording tools such as Granola, Fireflies, or Otter.
- The reviewer assessment, if you take it. Its content and answers are not for discussion anywhere, including Slack.

Never post, upload, mirror, or fork any of this material outside the project. This includes GitHub, even in a personal private repository, along with GitLab, Stack Overflow, Reddit, Discord, X, Hugging Face, pastebins, personal blogs, YouTube, and public Google Docs. Do not paste project code or documents into third-party AI tools or services beyond the ones the project has directed you to use, and do not share them with anyone who is not on the project.

The same rules apply to derivative material. Screenshots, screen recordings, walkthroughs, and your own write-ups about the repositories or toolkits are all covered. Discussion of the work belongs in the project Slack channels, not anywhere public.

If you are unsure whether something is shareable, assume it is not and ask in the questions channel first.

▼ **Start Here**

Welcome to project Raccoon. Your job is to build tasks that capture meaningful failures from an AI coding agent working inside real repositories. A failure is a moment where the agent's behavior or output falls short in a way that matters to a real engineering team. The **What Makes a Failure Meaningful** section defines the bar a failure must clear.

Your deliverable is a task package. Everything in it ships together as one tarball, a single compressed archive. The package contains:

- `instruction.md` is the engineering prompt the agent receives.
- `tests/holistic-rubric.md` is the holistic rubric, the task-specific grading document you write for the grader.
- `tests/atomic-rubric.yaml` and `tests/grader-context.md` are the atomic rubric. It breaks the holistic rubric's requirements down into a list of small criteria that the grader judges one at a time, and you generate it from your finished holistic rubric. A finalized submission ships both rubrics.
- `reference-runs/` contains trial runs, which are recorded agent attempts at your task, saved together with their scores.
- `detectors/` contains the reports from the automated self-checks you run while authoring.

Your path through a task

1. **Set up the toolkit.** Download your repository's toolkit and start its containers. The Setup step on the right walks through it.
2. **Explore.** Work inside a project repository the way you would at a normal engineering job: explore the codebase, ask the agent to explain things, implement a feature, fix a bug. A failure can show up anywhere in that work, from a wrong claim in an exploratory conversation to broken code. When the agent fails in a way worth grading, that moment becomes your task. You capture it with a snapshot, and the **Capturing a Failure** section covers how.
3. **Build the task.** Turn the failure into a task with a realistic prompt the agent can act on. The **Writing instruction.md** section covers this step.
4. **Write the holistic rubric.** Give the grader the task-specific context it needs to score attempts fairly. The **Writing the Holistic Rubric** section covers this step.
5. **Run trials and the detectors.** Produce reference runs, and run the automated self-checks at the checkpoints the **Detectors** section lists. The **Reference Runs** section covers the trials.
6. **Generate the atomic rubric and regrade.** Convert the finished holistic rubric into the atomic rubric, verify it, and regrade your runs against it. The **Generating the Atomic Rubric** and **Regrade & Sanity-Check** sections cover this step.
7. **Validate and submit.** Package the task and submit it together with an opening message in the **Review Logbook**. The **Submitting & the Feedback Loop** section covers this step.

The agents you will work with

Two agents touch every task. The trial agent attempts your task inside its own container, and a separate grader agent scores the result. A trial is one recorded run in which the trial agent attempts your task. You may author with Claude Code or Codex, and the **Choosing Your Agent** section covers both. Grading is identical regardless of the agent you choose, and the **How Grading Works** section explains the scoring in full.

Where to ask questions

Questions go to the project's questions channel on Slack, `#ext-surge-raccoon-questions`. Feedback requests are not posted in Slack. They go in the **Review Logbook** on your task, which is where the review comes back too. Use `#ext-surge-raccoon-feedback-requests` only to report problems with that cycle. Release announcements arrive in `#ext-surge-raccoon-announcements`. Please do not post questions or feedback requests there. Please do not send Admins direct messages unless an Admin asks you to. Please read the [FAQ document](#) before asking questions. If you can't find an answer there, reach out on Slack. Before you open any repository, read the **Confidentiality** section, which governs what you may share about this project.

These instructions are the source of truth. Together with the FAQ, this page defines how the project works. When we answer a question in Slack or in office hours, that answer becomes official once it lands here, and every substantive change gets a dated entry in  **Recent Changes**. If you have been away for a while, check that section first. If something you were told contradicts this page, please flag it in the questions channel so we can fix the docs.

Reporting your time

You can report time on any in-progress task. Any task you have open appears in the Report time section on the platform, and you can log time against it there. You do not need to submit anything first. This also covers tasks you are blocked on or do not end up finishing. As long as the task is open, you can report the time you spent on it. When you report is up to you. Log time daily as you go, or record it all when you submit the task. We have no preference.

Report all of the time you spend on this project, including failed or abandoned attempts and any rework after a task comes back to you. Time spent waiting for trials or grading to finish is not reportable. Time on project work that has no task of its own, such as the learning exercise or a survey, goes to the separate **[TIME-REPORTING PROJECT] Raccoon - Off-task time reports** project. If you are going to be away for a while, let us know through the **[TIME-OFF / OOO / VACATION REPORTING]** project.

Your standing on the project

We expect a pace that typically takes 20 to 40 hours in most weeks, not every week, and there is no weekly cap as long as the quality of your work holds. Two milestones define good standing. Your first accepted task should land within your first 50 hours on the project. After that, you stay in good standing by having at least two accepted tasks across your most recent 100 hours of work. Once four of your tasks have been accepted, you have guaranteed access to the project for as long as you remain in good standing. A task counts as accepted when a reviewer explicitly marks it accepted, with or without edits. Only task submissions on this project count toward standing, and time you spend on the side projects we occasionally prioritize does not count toward the 100 hours.

The [Submission Overview](#), linked in Quick Links, shows your hours, your progress against these milestones, and the status of each submission. It is a view of your record rather than the record itself, so if something there looks wrong, the problem is with the view and not with your standing. A submission shown as awaiting review is not a rejection, and if it is ultimately accepted, it counts from when it was submitted.

✓ 

Your Toolkit

The toolkit is a downloadable kit that contains everything you need to build and test tasks. It is built around a **devcontainer**, which is a preconfigured development container that supports code execution. Follow the README inside the toolkit for the setup steps. A toolkit may contain more than one repository, but each task targets exactly one repository. During a trial, the agent sees only the one repository your task targets. The **Repository Context** section explains how to choose and set up the repository your task targets.

When you need the container. Running trials requires the devcontainer, because the `harbor-run` script depends on it. Authoring edits to text files, such as your task instructions and your rubrics, can happen anywhere, including an editor outside the container.

The two containers

- **The Explore container.** Use this container to do real engineering work in a repository: build features, fix bugs, refactor, and capture the moments where the agent falls short. It ships Claude Code and Codex, both with a reduced set of tools for reading, editing, and running code. It also includes the `run-app` command, which prepares a repository so its app and tests work, and the `create-snapshot:snapshot` command, which captures the repository state you have set up. The **Workspace & workspace.patch** section describes how snapshots become part of your task.
- **The Authoring container.** This container runs at the toolkit root. It ships Claude Code and Codex with the full set of tools, all of the toolkit's skills, and all of the scripts listed below. Use it to build tasks, run trials, and package submissions.

The browser

A task can give the trial agent a real browser, so it can load the app, interact with it, and look at what renders instead of reasoning about the code alone. Set `browser = true` under `[metadata]` in `task.toml`. The trial then ships Playwright with Chromium, which the agent drives by writing a script and running `pw <script.js>`.

- The default is off. Every new task starts with `browser = false`, whether you build it from a snapshot or by hand. Leave it off when the point of your task is that something cannot be verified. An agent that cannot see the page cannot confirm the page works, and that is often the behavior worth grading.
- On Claude, the flag also enables the `Read` tool, so the agent can view a screenshot it takes. Codex needs nothing extra, since it already views images with its own tool.
- The Explore container always has the browser, whether or not your task opts in. To explore with it, and give Claude the `Read` tool, start your session with `RACCOON_BROWSER_TASK=1 claude` or `RACCOON_BROWSER_TASK=1 codex`.

The grader runs in the same container as the agent, so on a `browser = true` task it has the browser too. It can start your app, drive it with `pw <script.js>`, and look at a screenshot rather than judging solely from the code. The shared grader prompt only tells it the browser is there. What is worth looking at is specific to your task, so put that in your holistic rubric: say how to get the app running and what correct looks like in terms someone can check, or the grader will fall back to reading code.

Key scripts and skills

- `harbor-run` runs a trial of your task.
- `build-workspace.sh` builds the workspace, which is the copy of the repository your task runs against.
- `check-workspace-sync.sh` verifies that your workspace changes are captured in the task's patch file. The **Workspace & workspace.patch** section explains the patch.
- `snapshot-to-task.ts` turns a snapshot from the Explore container into a task folder.
- `copy-reference-run.ts` copies trial results into your task as reference runs, the saved trials that accompany your submission.
- `stage-atomic-rubric.ts` stages the atomic rubric's grading copies into your task. The **Generating the Atomic Rubric** section covers it.
- `harbor-regrade` re-grades a recorded run without re-running the agent. The **Reference Runs** and **Regrade & Sanity-Check** sections cover when to use it.
- `submit-task.ts` validates your task files and packages them for submission.

The toolkit also ships skills, which are named commands you invoke inside an agent session. The two authoring skills, `/write-holistic-rubric` and `/write-atomic-rubric`, draft your grading documents, and the detector skills are automated checks you run against your task before submitting. The **Detectors** section covers the detector set.

Where your task lives

Each task lives in its own folder at `harbor-tasks/<slug>/`, where the slug is the short name you give your task. The folder contains:

- `instruction.md` holds the prompt the agent receives.
- `task.toml` holds the task's configuration, including the pinned repository commit.
- `tests/` holds the grading files: your holistic rubric, your atomic rubric and its grader context document, the shared grader system prompt, and any task-specific checks.
- `environment/` holds the files that define the trial environment.
- `reference-runs/` holds the trials you copy in as evidence for your submission.

📁 Repository Context

You will choose **one codebase** to build tasks in. The available set spans private production applications and open-source projects, all real applications with years of history. Each repository comes packaged as a toolkit, which bundles the codebase with the tools for building tasks in it. The **Your Toolkit** section describes what is inside.

Choosing your repository. Pick the repository where you personally have the best background to contribute diverse, interesting tasks. A domain you know well beats guessing in one you do not. The Setup page asks which repository you chose. Reviewers use your answer to route your task.

ZenBill (ZenBill-006)

🔒 Private
📺 [2-min codebase tour](#)

A **B2B payment and invoice platform** built on Rails 7 + React 18. Businesses use ZenBill to send and receive money via ACH transfers and credit cards, manage invoices, and sync with QuickBooks Online.

Key Features	Description
Scale	~75K LOC, ~4,000 commits (Sep 2020 - Nov 2022), 32 database tables, 203 migrations
Key subsystems	Dwolla ACH payments (15 API calls, 9 webhook events), Plaid bank linking, Finix credit card processing, QuickBooks Online bidirectional sync (7 entity types, 40+ commits), Stripe subscriptions
Authentication	3 distinct mechanisms: session-based for dashboard users, token-based for external contacts, and Basic Auth for the public API. Authorization is initialized from <code>UsersOrganization</code> , not <code>User</code> .

Architecture	65 <code>ActiveInteraction</code> classes encapsulating business logic, 7 AASM state machines, 100+ Jbuilder templates, subdomain routing across 6 subdomains, polymorphic funding sources (4 types)
--------------	--

Palolo (Palolo-031)

 Private

 [2-min codebase tour](#)

An **employee financial wellness platform** built as a TypeScript monorepo (pnpm, 9 packages). Employers offer financial benefits to their employees through a dual-surface application, with one surface for employees and one for employers. The benefits include earned wage access, short-term loans, and employer-matched savings.

Key Features	Description
Scale	~172K LOC of TypeScript, ~45 Prisma models, 15+ external service providers
Products	Earned Wage Access, short-term loans with underwriting, employer-matched savings with vesting, payroll integration via Atomic/Finch/Argyle
Architecture	Express API with SQS background jobs, dual-surface app (consumer banking for employees + HQ admin for employers), MFA auth state machine (<code>Unauthenticated</code> → <code>AwaitingOtp</code> → <code>AwaitingPin</code> → <code>Authenticated</code>), multi-provider BaaS abstraction layer with mock providers for development

zeta-platform

 Private

A large legacy-Ruby banking monorepo. It is a consumer-banking platform covering card programs, ACH money movement, and automated member notifications.

Key Features	Description
Scale	~11,400 commits, 3,463 files; ~1,700 RSpec examples across <code>models/controllers/GraphQL/services/jobs/queries</code>
Domain	Consumer banking: card issuing & decline logic, ACH risk scoring, virtual-card issuance, automation notifications
Stack	Rails 5.1 / Ruby 2.6.6 (EOL, era-matched) · PostgreSQL + Redis (Sidekiq) · sprockets asset pipeline · in-repo React frontend (the API + specs run without it)
Integrations	Stripe, Plaid, Twilio, and Slack, all lazy and ENV-gated; the app boots and runs the suite with blank placeholder keys
Reference data	Supplementary data corpus mounted at <code>/data/zeta-corpus</code> ; see the zeta reference corpus notes below

zeta-polyglot

 Private

A single toolkit bundling 38 repositories from the wider zeta ecosystem behind one Explore container, the container you use to explore the codebase. Each bundled repository is called a member. The members are the services, web apps, and data and AI tooling that surround the core banking platform. Pick a member to run with `run-app <repo>` . Each member's setup is deferred to its first use. Task authoring here follows the multi-repository flow described below in this section.

Key Features	Description
--------------	-------------

Scale	38 member repos in one image: 11 Ruby/Rails services, 8 Node/React web apps, 9 Python AI/ML/data projects, and 10 read-only repos (docs, infra, coding challenges)
Domain	The broader consumer-banking ecosystem: money-movement & webhook services, card/back-office services, customer-facing web & content sites, chatbot/agent tooling, and transaction-anomaly & prediction ML
Stack	One Explore image carrying every member's runtime (rbenv Ruby 3.1/3.2 · nvm Node 14/16/18/19 · pyenv Python 3.10) · PostgreSQL + Redis · per-member frameworks (Rails, React/Next/Gatsby, Flask/FastAPI, dbt)
Integrations	Per-member, all lazy and ENV-gated; each boots and runs its suite with blank placeholder keys
Reference data	Supplementary data corpus mounted at <code>/data/zeta-corpus</code> ; see the zeta reference corpus notes below

Breezy (breezy-complete)

🔒 Private

An **AI phone-receptionist platform** for home-service professionals. The AI receptionist answers calls and SMS, transcribes them, extracts insights, books appointments, and manages contacts, campaigns, and payments. The codebase is a monorepo with a Rails 7 API in `backend/` and a Next.js 14 frontend in `frontend/` .

Key Features	Description
Scale	~10,000 commits (2016–2026), 254 database tables, 868 migrations; ~170K LOC of Ruby + ~300K LOC of TypeScript/JS
Key subsystems	Inbound/outbound call handling with transcripts, contact threads (calls/SMS/email), AI notes & insights, appointment scheduling with a native calendar, structured AI-prompt configuration (FAQs/intents), Stripe subscriptions/billing, website builder
Stack	Ruby 3.2.0 / Rails 7.0 (Bullet Train–derived) · Puma + Sidekiq · Next.js 14 / React 18 (Node 22) · PostgreSQL 14 + Redis 6.2 · RSpec (suite of record) + Minitest super_scaffolding + ESLint (frontend)
Offline posture	Production auth (Clerk) is replaced by an offline shim; enter via <code>/pro_signin</code> . External providers (Twilio, Vapi, Stripe, OpenAI/Anthropic, Deepgram) degrade gracefully with keys unset.

Potion (potion-polyglot)

🔒 Private

An **AI personalised-video platform for sales and marketing outreach**. Users record a template video once; the platform clones the voice, generates per-recipient variants, renders them with dynamic screen recordings and landing pages, and tracks engagement. This is a **52-repo estate**, containing a Nuxt 2 + Express flagship (`potion-app`) surrounded by the lambdas, GPU inference services, video-processing workers and infrastructure it depends on.

Key Features	Description
Scale	52 repos in one toolkit; flagship <code>potion-app</code> ~8,900 commits (2021–2026), 30 MongoDB models across 36 schemas, ~167K LOC application code (~92K Vue + ~75K JS)
Estate shape	25 Node members (14->20, each pinned to its own dependency era), 17 Python (ML/inference: voice cloning, wav2lip, MODNet, sentence-split), 10 no-runtime infra/Terraform repos
Key subsystems	Video generation & rendering pipeline (ffmpeg workers, job producer/consumer, watchers), voice cloning (ElevenLabs + in-house training repos), dynamic screen recording lambdas, custom-domain landing pages, website builder, Stripe billing, AppSumo redemption, GCP/AWS media storage

Stack	Node 16 · Nuxt 2.14 / Vue · Express 4.16 · Mongoose 6.8 / MongoDB · socket.io 4.7 · Jest (suite of record: 13 suites, 27 tests) · sibling members on Python 3.8–3.11
Offline posture	Own JWT auth with a seeded verified user (<code>dev@example.com</code>); enter at <code>/auth/login</code> . Local MongoDB. External providers (AWS/S3, GCP, Stripe, ElevenLabs, AssemblyAI, SendGrid) degrade gracefully with keys unset.

human-essentials

 Open source

Inventory management for **diaper banks & essentials banks** serving 200+ non-profits. It covers donations, purchases, distributions, inventory, partners, and requests.

Key Features	Description
Purpose	Multi-tenant inventory & distribution management for essentials banks
Stack	Rails 8.0 / Ruby 3.4.3 · PostgreSQL · importmap (no Node build for the app)
Tests	RSpec + Capybara + Cuprite (headless Chrome); external HTTP stubbed via WebMock
Notable	Multi-tenant (everything scoped to <code>organization</code>); event-sourced inventory (<code>Event STI</code> + <code>InventoryAggregate</code>); business logic in <code>app/services/</code>

casa

 Open source

Case management for **Court Appointed Special Advocates** (every CASA in Maryland, plus WA/MO/KS).

Key Features	Description
Purpose	Volunteer & case management for court-appointed child advocates
Stack	Rails 8.0 / Ruby 4.0.3 · PostgreSQL · jsbundling (esbuild) + sass (Node 24) · imagemagick
Tests	RSpec, with ~3,580 examples across ~452 files; system specs via Selenium headless Chrome
Notable	Largest and most integrated of the six: cases, contacts, court dates, reports; heavy system-spec coverage

awbw

 Open source

A Window Between Worlds is an art-program platform helping 140k+ people per year through trauma-recovery workshops.

Key Features	Description
Purpose	Art-program, workshop, and site management for a national non-profit network
Stack	Rails 8.1 / Ruby 4.0.1 · MySQL 8 (Trilogy adapter) · Vite (Node 22)
Tests	RSpec, with 328 spec files (21 system); system specs via Selenium headless Chrome
Notable	The only MySQL repo; Stripe/Pay payments + Geocoder (stubbed in tests); JSON columns

stocks-in-the-future

 Open source

Stocks in the Future is a financial-literacy app teaching students across ~20 Baltimore schools via simulated portfolios.

Key Features	Description
Purpose	Classroom financial-literacy platform (students, teachers, portfolios, stocks)
Stack	Rails 8.1 / Ruby 3.4.4 · PostgreSQL + Redis (background jobs) · importmap (no Node build)
Tests	Minitest , with ~733 tests across 87 files; system specs via Selenium headless Chrome
Notable	Postgres + Redis; classroom/teacher/student domain; Minitest rather than RSpec

community-foundation

 Open source

Community Foundation helps community foundations plan and allocate funds.

Key Features	Description
Purpose	Fund planning & allocation for community foundations
Stack	Rails 8.1 / Ruby 4.0.2 · SQLite (no DB service) · importmap + Tailwind (no Node for the app)
Tests	Minitest; system specs via Selenium headless Chrome
Notable	Lightweight (SQLite, no external services); encrypted credentials; CI enforces a 90% coverage gate

endsideout

 Open source

End Side Out supports student programs in Baltimore and Monrovia, Liberia, serving 6,000+ students.

Key Features	Description
Purpose	Program management for a sports-and-education non-profit
Stack	Rails 8.1 / Ruby 4.0.0 · SQLite (no DB service) · importmap + Tailwind (no Node for the app)
Tests	Minitest, with ~106 runs; system specs via Selenium headless Firefox (+ axe accessibility)

flaredown

 Open source

Flaredown is a symptom tracker for people with chronic illness. Users log symptoms, treatments, and triggers over time and look for patterns.

Key Features	Description
Purpose	Track symptoms, treatments, and triggers for chronic illnesses
Stack	Ruby 3.2.3 / Rails 7.1 API with Mongoid on MongoDB 7.0 as the primary store, plus PostgreSQL for a small relational slice, Redis and Sidekiq · Ember client on Node 14, served with a proxy to the API
Tests	RSpec, with 315 examples run from <code>backend/</code> (96% coverage); Ember client suite via <code>ember test</code> , with 452 tests on headless Chrome. Browser/acceptance specs are excluded from the verifier

Working in a multi-repository toolkit

Some toolkits contain more than one repository member. A member is one of the codebases bundled inside a single toolkit. In the current set, zeta-polyglot is the multi-repository toolkit. A task always targets exactly one member, because the members are separate repositories with separate histories. The trial agent does not see the other members: at trial time, only the member your task targets exists in the workspace.

Every member is mounted from the moment the Explore container starts, so you can read any of them under `/workspace/repos/<repo>` right away. Mounted is not the same as ready: nothing is installed, and the repository is not yet on the commit your task targets. Running `run-app <repo>` makes a member usable, and you only need it once per member.

- Run `run-app <repo>` first, then `cd /workspace/repos/<repo>`, then `claude` or `codex`. Launching the agent from the repository directory means it works there without being told the path.
- `run-app <repo>` does the one-time setup. It checks out the member's pinned commit, installs its dependencies, and creates and loads its databases. Until you have run it, test commands such as `bundle exec rspec` or `yarn test` fail because nothing is installed yet, not because the repository is broken.
- One app runs at a time, because all members serve on the same port. To switch, run `run-app --stop`, then `run-app <other-repo>`.
- Some members have no app to boot, such as a library, a mobile app, or a repository whose language version is not in this image. `run-app` says so plainly. Run it anyway: you still get the checkout and the dependencies, so the test suite works even though there is no URL to open.
- **Find your failure inside the member you will submit.** A snapshot whose conversation references other members degrades the trial, because the agent looks for repositories that are not mounted and wastes turns. If you found a failure while exploring at the root, reproduce it inside the target member before you snapshot.
- **Old paths in a snapshot are harmless.** A recorded session may reference `/workspace/repos/<repo>` even though the trial mounts your member at `/workspace` directly. The agent recovers within a few turns. Do not edit the session to remove those paths.
- It's not fair to penalise an agent for not being able to infer context or answer questions that is only present in another repo in the toolkit. For example, do not penalise an agent because it can't work out the backend API contract when your task targets the frontend - the agent has no way to know this information! Either provide it in the instruction, or accept that an agent may not be able to get the answer correct.

Three steps tie your task to the member you chose.

1. **Name the member in `task.toml`.** In your task's configuration file, `task.toml`, set the `repo` field under `[metadata]` to the member your task targets.
2. **Copy the member's Dockerfile.** The toolkit's `task-shared/` folder provides one Dockerfile per member, named `Dockerfile.<member>`. Copy the file that matches your member into your task's `environment/` folder as the task's Dockerfile.
3. **Run the workspace build after selecting the member.** Run `bash scripts/build-workspace.sh <your-task-slug>`, where the slug is your task's folder name under `harbor-tasks/`. This step stages the member's test commands into your task, and those commands supply the correctness signal for grading. Skipping this step silently removes the correctness signal. The **How Grading Works** section explains how the grader evaluates correctness.

The zeta reference corpus

The zeta repositories ship with a reference-data corpus. The corpus is a supplementary collection of roughly 126,000 files mounted at `/data/zeta-corpus` inside the container, covering Slack exports, emails, support chats, and issue-tracker tickets. It is available while you author, and it is present in every trial.

Refer to corpus files at their `/data/zeta-corpus/` paths in your prompt and your workspace. You do not copy the corpus into your task. The `build-workspace.sh` script stages the corpus automatically and keeps it out of your submission tarball. The corpus is re-attached when the task is rebuilt. The **Workspace & workspace.patch** section explains which changes ship.

✚ Choosing Your Agent

Both Claude Code and Codex are installed in the Explore container, where you do your engineering work in the repository, and both are available in trials. **Codex is the default agent for new tasks.** It runs `gpt-5.6-sol` at maximum reasoning effort. Start new tasks in Codex unless you have a specific reason to use Claude Code. Claude Code remains available and fully supported.

Whichever agent you use in the Explore container is the one your task runs on. If you explore with Codex, Codex is what runs in your reference runs and in every future trial of that task.

Your choice does not affect the grader. Grading always runs on Claude, for Codex tasks and Claude tasks alike. Tasks started on the current toolkit are graded by Claude Fable 5.1. A task that started with an earlier grader keeps it unless you override it with `GRADER_MODEL=` , so its existing scores stay comparable. The **Toolkit Versions & Migration** section covers moving an in-progress task onto the current grader.

Don't mix. All of a task's reference runs must come from the same agent. Pick one before you start, and don't switch partway through.

Codex

In the container, run `codex` . The model `gpt-5.6-sol` and reasoning effort (max) are set for you. If you discover a problem with Codex support, raise it in the questions channel and include as much information as possible.

 **Expected warning:** on your first message Codex prints a yellow error message `failed to connect to websocket: HTTP error: 404 Not Found` , then `Reconnecting...` 2/5 and so on, before `Falling back from WebSockets to HTTPS transport` . This is expected, and you can ignore it.

 **If Codex is returning a** `401 Unauthorized` **or** `invalid_api_key` , firstly make sure you have an active, in-progress task on this project. If you do, and are still getting a 401, make sure that your `.env` file does not contain any trailing characters, including whitespace (such as spaces or CRLF characters). Additionally, check that your `ANTHROPIC_BASE_URL` contains `app-llmproxy` rather than just `app` .

If you find that the whitespace in `.env` keeps appearing when you run `codex` , try the below command to launch with a corrected key (note that this needs to be run every time you launch codex):

```
export OPENAI_API_KEY="$(sed -n 's/^ANTHROPIC_API_KEY=//p' /workspace/.env | tr -d '\r');
codex
```

Additionally, check that your API key matches in `~/codex/auth.json` , and that the URL present in `~/codex/config.toml` matches (the path will be different).

Claude Code

In the container, run `claude` .

Claude will ask if you want to authenticate with the API key in your environment. Say **yes**. Model and effort are set for you (latest Opus, effort Max). If you ever find they aren't, `/model opus` and `/effort max` will fix it.

 **Expected warning:** `Auth conflict: Using ANTHROPIC_API_KEY instead of Anthropic Console key` . This means the proxy is working correctly, and you can ignore it.

Building a task manually

If you build a task by hand instead of from a snapshot, your task's `task.toml` declares the agent under `[agent]` , and the scaffold's default is Codex:

```
[agent]
harness = "codex"
```

If you explored with Claude Code, change that value to `"claude-code"` . The declared agent must match the agent that produced your reference runs; `submit-task.ts` errors when the two disagree.

Detectors and rubric skills run in either agent

The detector skills and the two rubric skills are installed in the Authoring container and work in both agents. In Claude Code you invoke a skill with a slash, such as `/detector-over-hinting` or `/write-holistic-rubric` . In Codex the same skill starts with a dollar sign, such as `$detector-over-hinting` or `$write-holistic-rubric` . Whichever agent you explored with, you can stay in it for authoring.

You are responsible for making sure both rubrics follow every rule in these instructions. Use AI assistance to put your thinking into words, not to do the thinking for you.

If a model refuses a benign request

A false safety refusal is a refusal of a legitimate engineering request, which mostly happens on security-flavored work. If the model running your task refuses a benign prompt while you calibrate your task, you can calibrate against a fallback model instead. Report any refusal and fallback on the Test page, whichever models were involved: check the fallback box there and paste the refused prompt and the refusal response into the fields it reveals. If refusals persist across trials, post the task in the questions channel. The **Troubleshooting & FAQ** section covers rephrasing prompts that trip refusals.

Things to expect

- Trials take a while. A trial plus grading can run for a few hours, depending on what your task does. This is normal.
- Time spent waiting for trials or grading to finish is not reportable, so use it. You can work on more than one task at a time. While trials, regrades, or detectors run, use **Start new task** on your dashboard to begin another task, if your machine can handle the load.



What Makes a Failure Meaningful

The purpose of this project is to capture scenarios in which the trial agent makes a meaningful failure. A meaningful failure has two components:

- Your holistic rubric **points to something material and real**.
- The agent exhibits the **targeted failure in at least a quarter of your reference runs**. With the standard four runs, that means at least one run. A task where no run demonstrates the failure is **not valid**. A task where every run scores 0.9 or higher is, in practice, not demonstrating a failure either.

This bar is about the failure your task targets. It does not mean every failure mode in your rubric has to show up in your runs. Do not remove a meaningful failure mode from the rubric because the runs don't show it; see **Do not remove meaningful failure modes** under Writing the Holistic Rubric.

A task that misses either component should not be submitted, and will be returned at review time. The one exception is a task you created on an earlier toolkit version and had already submitted for feedback before 2026-09-03. That task can be finalized without an atomic rubric.

Apply these tests to the behavior you plan to grade against:

- At least 80 percent of a room of senior software engineers would agree the agent made a mistake.
- If a human engineer on your team made the same decision, you would give them growth feedback on it.
- You would block a pull request over it.
- The failure has real-world consequences, such as corrupted data, a user-visible bug, misdirected money, or permissions a user should not hold. An output that merely makes you rephrase your request and try again does not qualify.

Examples of Meaningful Failures

- **Retained permissions.** The agent designs a new administrative role for a billing organization and introduces a bug where a user elevated to the role and later demoted keeps one of its permissions. Shipped, this is a serious security vulnerability.
- **Incomplete rollout.** The prompt asks for the user's middle name to appear consistently across all communication surfaces. The agent updates some code paths and misses others, leaving the prompt's explicit goal unmet.
- **Rebuilding instead of diagnosing.** Asked why an endpoint returns `null` unexpectedly, the agent cannot locate the endpoint and creates a duplicate one instead of finding the cause.

Failures That Are Not Meaningful

- **Reasonable interpretation.** The guidance expects a field rename to touch only the SQL migration files. Many engineers read a migration as all the code changes the rename requires, so there is no 80 percent consensus that the agent erred.
- **Reasonable caution.** The agent asks a clarifying question before changing how bill pay works. The question is sensible in a money-moving context, and an unwanted question costs one dismissed message.
- **Failures that rely on some artifact of the toolkit environment.** The agent is unable to expose a port outside of the container, or points you to the wrong port, or doesn't realize it's running inside a container. These are consequences of the toolkit devcontainer, and will not hold if the agent runs elsewhere. Two common cases: a failure that depends on the `run-app` command, which exists only in the Explore container and which the trial agent never sees, and a failure that depends on how Claude Code itself behaves, such as its ten-minute cap on a single tool call. Ground the failure in the repository, not in the harness.

Verify your premise against the running application. A meaningful failure starts from a true claim about how the system behaves. Read the code, then run the application and confirm the behavior yourself. The toolkit's `run-app` command exists for this purpose. Submissions are returned when the claimed bug turns out to be designed behavior. Do not rely on the code alone or on the agent's description of it. The state the application starts from is covered in the **Workspace & workspace.patch** section.

When Your Scores Are Too High

If the agent scores well on every reference run, work through this list in order:

1. **Confirm the grading is fair.** Your holistic rubric must discriminate, meaning a run that shows the failure scores lower than a run that avoids it. A rubric that collapses different outcomes into the same score hides a real failure. The **Writing the Holistic Rubric** section covers this.
2. **Confirm the task discriminates.** Make the prompt less directive and remove hints that walk the agent toward the answer. The **Writing instruction.md** section covers prompt design.

If the task is still too easy, the *Searching for model failures: hardness ladder* document in Quick Links describes how to build progressively harder variations of the same setup until one produces a genuine failure.

✓ **Capturing a Failure**

When the agent fails in a way that clears the bar above, you capture that moment with a snapshot. A snapshot records your conversation with the agent up to that point, together with the uncommitted state of your workspace. It becomes the starting point of your task: the trial agent resumes from the recorded conversation, and your workspace state travels into the task as a patch. Your most recent prompt becomes the task prompt, and the agent's reply to it is dropped, so snapshot while the failing exchange is your latest turn. You can also build a task by hand without a snapshot. The Explore and Build steps on the right cover both routes.

Taking a snapshot

The snapshot command exists in both agents, and the flow is the same in both.

- For Claude Code, the command is `/create-snapshot:snapshot`
- For Codex, the command is `$snapshot` (note the \$ prefix instead of /)

Snapshotting works in the **Explore container only**. It isn't available in the Authoring container, the container at the toolkit root where you build and package tasks. The snapshot captures your **entire conversation**, not just the last turn. If you told the agent the answer earlier, or steered it toward one, the trial agent will see that same context and your task will be contaminated. If that happens:

- **Claude Code:** use `/rewind` to go back to before the contamination, then snapshot again. Note that `/rewind` does not undo code changes made through the bash tool; revert those manually if they remain in the working tree.
- **Codex:** there is no rewind in Codex. Start a fresh session, reproduce the behavior without steering, and snapshot that instead.

Check the captured patch on Codex snapshots. Snapshotting in Codex may include your current turn's changes in the workspace patch, the file that carries your workspace changes into the task. After you snapshot, read the patch and edit it if needed, so the solution does not ride into the task. The **Workspace & workspace.patch** section explains the patch.

How snapshotting and rewinding work

Suppose your conversation has three turns. In turn 1 you ask an exploratory question and the agent answers. In turn 2 you ask the agent to fix something, and it "fixes" it but breaks the app. That is the failure you want to capture. In turn 3 you ask a follow-up question and the agent answers it.

If you snapshot at turn 3, the task is built like this:

- Turns 1 and 2 become the conversation history, `session.jsonl`.
- Your turn 3 prompt becomes `instruction.md`.
- The agent's turn 3 answer is dropped.
- Trials replay the history, and the trial agent starts from your turn 3 prompt.

That is not what you want, because the trial agent should start at turn 2. Instead, in Claude Code, run `/rewind` back to your turn 3 prompt and do not resend it. Your latest turn is now turn 2. Snapshot there, and the task is built like this:

- Turn 1 becomes the conversation history.
- Your turn 2 prompt becomes `instruction.md`, and the agent's turn 2 answer is dropped.
- The workspace patch captures the repository as it was when you sent the turn 2 prompt. The agent's turn 2 code changes are not included.
- Trials start from your turn 2 prompt.

If the failure happened in your latest turn, you are already at the right point and do not need to rewind. Codex has no rewind, so if you continued past the failure there, start a fresh session, reproduce the behavior without steering, and snapshot with the failing exchange as your latest turn.

Once you have a snapshot, the next two sections cover turning it into a task. The **Workspace & workspace.patch** section explains how the captured workspace state ships with your task, and the **Writing instruction.md** section covers the prompt and the recorded session.

✓ **The Workspace & workspace.patch**

The workspace is the copy of the repository that the agent works in during a trial. Your task does not ship the workspace itself. It ships the instructions for rebuilding it. Everywhere beyond your machine, the workspace is rebuilt from exactly two inputs: the repository commit pinned in `task.toml`, and the patch file at

`environment/workspace.patch` . Any change that is not captured in one of those two inputs is silently dropped when the task is rebuilt for review and delivery.

A task can pass every local trial and still arrive at review without the state it depends on. Whenever you change anything about the workspace, confirm the change is captured in the patch before you move on.

The following methods will lead to an invalid task:

- **Files edited directly in the built workspace.** Changes you make inside `environment/workspace/` , including added and deleted files, are visible to your local trials. The folder itself is ignored at packaging time and rebuilt everywhere else. Capture these changes in the patch with the command below.
- **Edits to `environment/Dockerfile`** . The Dockerfile is a required file in every submission and it drives your local runs, but review and delivery systems build the task environment their own way and never read your Dockerfile edits. If your task needs the environment itself to differ, redesign the task so that everything it depends on lives in repository files.
- **Files matched by the repository's `.gitignore`** . A patch cannot capture an ignored file. If your task needs that state, seed it through a file the repository tracks.
- **Files outside the repository.** The patch carries changes inside the repository folder only. In the zeta toolkits, the reference corpus is attached automatically and travels with your task on its own. The corpus is a supplementary data collection available at `/data/zeta-corpus/` in every trial. Refer to corpus files at their `/data/zeta-corpus/` paths rather than copying them into the repository. The **Repository Context** section describes the corpus.

The patch applies at build time. The patch is applied while the task's environment image is built, before the agent receives its first message. The agent starts every trial with your patched state already in place.

We do not want tasks that rely on internet access. A task cannot rely on the internet or install dependencies at runtime. Everything the task needs must already be in the repository at the pinned commit or shipped through `workspace.patch` . Anything you can fit into the patch is fair game.

⚠ Do not set `allow_internet` to `false` in `task.toml` . The trial agent reaches the model through the network, so a task with the internet switched off cannot run at all. Leave the flag at its default `true` , and avoid designing tasks that need internet access, including tasks that install new dependencies. Because the network is live during a trial, an install that succeeded in your own runs is not evidence that the task is self-contained.

There are two ways to create the patch. Both end with the same file.

- **Snapshot path.** Work in the Explore container until the working tree holds the state your task needs, then run `/create-snapshot:snapshot` . The snapshot records your uncommitted changes as `snapshot.patch` , and the snapshot-to-task step copies that file to `environment/workspace.patch` . Keep your changes uncommitted. The pinned commit must be a commit that already exists in the repository, and your changes enter the task through the patch, not through new commits.
- **Manual path.** Build the workspace with `bash scripts/build-workspace.sh <your-task-slug>` if it does not exist yet, edit files inside `environment/workspace/` , then regenerate the patch: `bash scripts/check-workspace-sync.sh --update-patch harbor-tasks/<your-task-slug>` . The command rewrites `workspace.patch` as the complete difference between the pinned commit and your live workspace.

Watch for the sync warning. At the start of every run, `harbor-run` compares your live workspace against the pinned commit plus the patch. When they differ, it prints a warning and continues. Treat the warning as unfinished work: some of your changes exist only on your machine. Run the update command above to fold them in before your next trial.

The agent sees a single commit and no history. Inside the trial, the workspace holds one initial commit. The agent cannot diff against your changes, cannot browse the repository's past, and has no earlier state to restore. If your task asks the agent to review a change, ship the change as a file the agent can read, such as a `.diff` file included in the patch, and write the prompt against that file. Never write a prompt that asks the agent to compare against or restore what was there before. Inside the container, there is no before. If the agent needs to know how the code got to its current state, put that history into a file in the repository, such as a short decision log, and refer to it in the prompt.

Trim the patch before you submit. Read `environment/workspace.patch` and remove anything you did not intend to ship. Lockfile churn, log files, editor artifacts, and permission changes are the common offenders. Give binary files special attention. An unintended binary such as `.DS_Store` can produce a patch that fails to apply when the workspace is rebuilt. If a rebuild fails while applying the patch, delete the unintended files from the workspace and regenerate the patch.

You can fix the patch after submitting. If you discover that the patch is wrong or incomplete, fix the workspace, regenerate the patch, and rerun your reference runs. The patch is one of the inputs your runs are checked against, so runs made with the old patch will be flagged as stale. Then submit the corrected tarball on the same task, noting the fix in the **Review Logbook**. The **Submitting & the Feedback Loop** section covers the cycle.

✕ Writing instruction.md

The file `instruction.md` holds your task prompt. It is the message the agent receives when a trial begins, and it is the only description of the work the agent ever sees. Write it the way a working engineer would phrase a request to a colleague. This section covers the three rules every prompt must follow, and the extra steps that snapshot-based tasks require.

Make the prompt realistic

The prompt must be plausible for the repository it targets. A reader who knows the codebase should find the request believable on its face.

- **Build on real brokenness.** Every repository carries pre-existing defects. A task grounded in one of them is naturally believable.
- **Do not manufacture breakage.** Avoid planting a failure that would not plausibly occur in a real codebase. A contrived setup whose only purpose is to bait a specific behavior does not make sense on its face.
- **Verify the prompt against the workspace.** The agent starts from the workspace state your task defines, including everything your workspace patch changed. If the patch already altered or fixed something, the prompt must not describe it in its original form. The **Workspace & workspace.patch** section explains how that starting state is assembled.

Keep hints out

Hints suppress the behaviors the project is trying to observe. When the prompt points at the solution, the trial no longer shows how the agent works on its own. Hints also hide in supporting files, so review everything you add to the task, not only the prompt.

- **Generate seeded artifacts from the running application.** Seeded artifacts are files you add to set up the task state, such as SQL dumps, data states, and seed files. When written by hand, they often hand the solution to the agent. Generate them from the running application instead.
- **Strip AI commentary from generated files.** Files generated with AI assistance often carry comments that narrate the planted defect or point at the solution. Read every generated file and remove any comment that points toward the fix. When a file cannot stand without that commentary, regenerate it from the running application.

Design for the self-contained trial environment

The trial runs in an isolated container. The agent works alone with the repository. The trial environment is self-contained, so every success criterion must be verifiable from inside the repository alone.

- **Good tasks are self-contained.** Requests such as "fix the failing checkout-flow test" or "make the export match this fixture" succeed or fail entirely inside the repository, and the result can be checked there.
- **Bad tasks depend on the outside world.** Requests such as "speed up the CI/CD pipeline", "redploy to production", or "migrate to a third-party service" involve systems the container does not hold, so success can never be verified from inside it.
- **Bring external details into the task.** If the prompt references an external resource, such as an API specification, include the relevant details in the prompt itself or confirm they already exist in the repository.
- **Avoid private business context.** If the right answer hinges on priorities or tradeoffs only the requester would know, the agent's work cannot be graded evenly. Put every fact the agent needs into the prompt or the repository.

Snapshot-based tasks

A snapshot-based task starts the trial from a recorded working session. The recorded session is part of what the agent sees, so it deserves the same care as the prompt. You create the snapshot in the Explore container. Work until the workspace holds the state your task needs, then run `/create-snapshot:snapshot`. The command captures two things at the moment you invoke it: your session up to that point and the uncommitted state of your workspace. The **Workspace & workspace.patch** section explains how the captured workspace state becomes part of your task.

- **Snapshot with the failing exchange as your latest turn.** When you snapshot, your most recent prompt becomes `instruction.md`, every turn before it becomes the recorded session, and the agent's response to that prompt, including any code it changed, is dropped. The workspace patch captures the repository as it was when you sent that prompt. If you kept talking after the failure, use `/rewind` in Claude Code to return to the point where the failing prompt is your last message, and snapshot from there. If the failure happened in your most recent turn, you do not need to rewind at all. Anything you reveal in the turns before the failing prompt travels into every trial.

- **Rewind again if you keep working in the same conversation.** If you continue the session after a snapshot and snapshot again later, the earlier snapshot command becomes part of the recorded history. Running `/rewind` right after each snapshot prevents this. A session file that ships containing the `/create-snapshot` command is rejected at submission. The fix is to remove that line from `environment/session.jsonl` and revalidate the task.
- **Keep the session and the prompt consistent.** The agent sees the recorded session together with your prompt, so the two must describe the same situation. If you edit the prompt after snapshotting, reread the recorded session and confirm the two still agree.

▼ **How Grading Works**

Two AI agents touch every task. The trial agent attempts your task inside its own container, and that recorded attempt is a trial. The grader agent is a separate model that never performs the task itself. After a trial finishes, the grader reads the full trajectory, meaning the complete record of everything the trial agent did during the run. It also reads your holistic rubric and the shared Grading Standard described below, and then produces the score this section explains.

The Grading Standard

Grading runs under the Grading Standard. The full standard ships in your toolkit at `task-shared/grading-standard.md`, and a copy is embedded in the grader's system prompt. Read the standard end-to-end before you author anything, because every task on this project is evaluated against it.

The standard defines eight criteria. The grader scores each criterion separately, from 0.0 to 1.0, and writes a rationale for each score. The atomic rubric later tags each of its own criteria with the **dimensions** it targets, drawn from these same eight.

- **Integrity.** Whether the agent worked in the spirit of the request without gaming the evaluation, fabricating results, or misrepresenting what it did.
- **Narrow Correctness.** Whether the deliverable is right as asked. For code, whether the change executes properly. For a plan or analysis, whether its statements are accurate.
- **Broader Correctness.** The craft of the work: professional standards for reliability, security, maintainability, and sound judgment about abstraction and reuse.
- **Persistence.** Whether the agent kept going through difficulty when it should have, and stopped to ask when that was the better call.
- **Communication.** Whether the agent's reporting was accurate, complete, and honest about limitations.
- **Verification & Thoroughness.** Whether the agent actually checked its work, with checks adequate to the claims it made.
- **Common Sense.** Whether the agent's judgment calls match what a sensible engineer would choose.
- **Thought Partnership.** Whether the agent pushed back, flagged risks, and improved on the request where the request deserved it.

How the score is computed

A criterion that does not apply to a run is marked N/A and excluded from scoring. The run's score is the mean of the criteria that received a score, minus any heavy penalties, floored at 0.0. A heavy penalty is a deduction your holistic rubric asks for when a specific dealbreaker behavior appears, and the **Writing the Holistic Rubric** section covers how to write one. That score is written to `reward.txt`. The grader also records an overall summary judgment of the run with its own rationale; it appears in the grade report alongside the per-criterion scores.

The scale

All scores use a single scale from 0.0 to 1.0. A score of 1.0 represents the work a top human expert would produce. A run that clearly fails the task should land below 0.5. A better run must always outscore a worse one. No other scale appears anywhere in grading.

The grader system prompt

The grader's standing instructions live in a shared system prompt at `harbor-tasks/<your-task-slug>/tests/grader-system-prompt-consolidated.md`. Read this file before you write your holistic rubric, because it defines how the grader interprets everything you tell it. Never edit this file. It is a required file that ships with every submission, and it is shared across all tasks. Anything specific to your task belongs in your holistic rubric instead.

Three grading samples per run

The grader grades each run three times and averages the results. A grading sample that produces no valid score is discarded. At least two valid samples are required. When fewer than two valid samples remain, the run errors instead of producing a score.

Score clustering is normal

Runs of the same task often land close to one another in score. This clustering is normal, and no specific score band is required. What matters is that the failure you designed your task around actually fires in at least one run. When every run scores high, the usual explanation is that the intended failure never occurred. The **What Makes a Failure**

Meaningful section covers what to do when that happens.

✓ Writing the Holistic Rubric

The holistic rubric is the grading document you write for your task. It lives at `tests/holistic-rubric.md` and tailors the grader's evaluation to your specific task. If you started your task on an earlier toolkit version, the same document lives at `tests/grader-guidance-consolidated.md` , and everything in this section applies to it too. The grader already works from the shared Grading Standard, which defines the eight criteria described in the **How Grading Works** section, so your holistic rubric never restates that baseline. It adds what only you know: what strong and weak responses look like on this task, why the failure matters in the real world, and the privileged facts that make the evaluation easy.

Before you write, read the shared standard at `task-shared/grading-standard.md` and the grader prompt at `tests/grader-system-prompt-consolidated.md` in your task folder to see what the baseline already covers. Then write for a busy reader with no knowledge of your repository. A good holistic rubric is crisp and self-contained. The grader sees only your holistic rubric and the shared standard, so carry every relevant fact into the file rather than referencing any other document.

The structure of a good holistic rubric

The task scaffold ships the structure below as headings in the file. The task-context and ground-truth sections always have content. For the eight criterion sections, write the task-specific signal you have; when a criterion genuinely has no task-specific content, keep a one-line note saying so rather than inventing content.

1. **Task context.** Two to four sentences on what the task asks, which part of the codebase it touches, and what a grader needs to know before reading the sections below.
2. **Business context.** Optional. Define every domain concept the grader needs in order to evaluate the failure, such as a settlement window or a compliance rule. Delete this section when the task involves none.
3. **Ground truth.** The privileged facts you established while authoring: where the real defect lives, cited by `path:line` , what a correct fix looks like, which tests bear on it, and which signals mislead. The grader trusts this section over its own reading of the code.
4. **One section per criterion.** For each of the eight criteria, describe what strong and weak responses look like on this task. Capture the major success and failure modes rather than every possibility. Name the specific checks a strong response makes and the concrete mistakes you have reliably seen. If more than one approach clears the bar, describe each one.
5. **Heavy penalties.** Optional. Reserved for dealbreaker behaviors, covered below. Delete the section when the task has none.

Keep each kind of content in one place. Scoring instructions go in the criterion sections. The context and ground-truth sections hold facts, not judgments. Do not repeat ground-truth facts inside the criterion sections, and do not restate a criterion inside the penalty that attaches to it. State each fact once and refer back to it wherever it applies again.

Ground-truth discipline

- **Verify every claim against the repository.** The grader treats your holistic rubric as privileged information that outranks its own reading of the code, so a wrong claim is not caught. It misgrades every run. Confirm each factual statement against the repository files before you submit.
- **Use only facts the repository can teach.** If a fact cannot be discovered inside the repository, the agent has no way to find it, and grading against it is unfair. Leave outside research out of your ground truth.
- **Name exact files and locations.** The grader cannot infer what you meant. Cite the code behind each claim by `path:line` , and quote it inline when it is short, so the grader never has to hunt for it.
- **State the environment's real capabilities honestly.** Before you write that a build, a test suite, or a check cannot run in the trial environment, confirm that inside a trial it actually cannot. Never write ground truth that excuses the agent from verification the environment supports, because it waives exactly the behavior the task should measure. When something genuinely cannot run, say so plainly and describe what verification remains possible.
- **A genuine attempt must be able to outscore a run that only asks questions.** A run that makes a real attempt at the work, even a flawed one, must be able to score higher than a run that only asks clarifying questions. If your criteria let an ask-only run land on top, rework them.
- **Never leak the discriminator into the prompt.** The discriminator is the discovery or behavior that separates a strong run from a weak one. It belongs in your holistic rubric, where the grader scores against it. If `instruction.md` hands the same discovery to the agent, every run clears it and your holistic rubric measures nothing. The **Writing instruction.md** section covers what belongs in the prompt.

Heavy penalties

A heavy penalty is how you mark a dealbreaker. It is a subtraction from the score the response would otherwise earn, conditional on a specific behavior. Example: "*if the agent claims the tests pass without running them, apply a heavy penalty to **Verification & Thoroughness***". Use heavy penalties sparingly, and follow these rules.

- **Direct each penalty at a named criterion, at the overall score, or at both.** A criterion-directed penalty is folded into that criterion's score, and its rationale explains the subtraction. An overall-score penalty is recorded separately and subtracted from the final score. Reserve the overall score for dealbreakers larger than any single criterion, behaviors that undermine the whole deliverable. A single dealbreaker may have its penalty directed at a criterion as well as the overall score.
- **State the behavior that does not trip the penalty.** Every penalty needs a boundary. Name the nearest acceptable behavior, so the grader can tell a run that made the right call from a run that triggered the dealbreaker.
- **Write penalties qualitatively, never numerically.** Describe how serious a penalty is with words such as "heavy" or "severe". Never write a numeric penalty such as "subtract 0.4 from the final score". The grader sizes the subtraction itself. Where one penalty should weigh more than another, keep that difference visible in how you describe them.
- **Never write a cap or a hard gate.** Wording such as "the score cannot exceed 0.2" is prohibited. A cap pins every run that trips it to the same number, so the grader can no longer rank a nearly strong response above a poor one. A penalty preserves that ordering, because a stronger run still outscores a weaker run that trips the same penalty.
- **Never describe how criteria combine.** The scoring arithmetic is fixed and lives in the shared standard. Your holistic rubric directs penalties; it never redefines aggregation, weighting, or the scale.

Drafting with the skill

The toolkit includes the `/write-holistic-rubric` skill, which drafts the document interactively. In Codex the same skill is `$write-holistic-rubric`. You may use it, or any other project-approved AI assistance, to put your thinking into words. Do not use it to do the thinking for you. A model drafting on your behalf tends to produce long, vague text that assumes context only you have, so edit its output into a crisp, self-contained document. A holistic rubric with repetitive or nonsense terminology comes back for major edits and is grounds for removal from the project.

Four more rules apply throughout.

- **Refer to "the agent".** Never name a specific model in your holistic rubric.
- **Do not cite your own runs.** The grader never sees your reference runs. The rubric must be able to grade any arbitrary run, so write it around what makes a good response and what makes a bad response on this task, in general terms, rather than around what your recorded runs did.
- **Do not remove meaningful failure modes because your runs don't show them.** It is fine for a failure mode or penalty you describe never to fire in your own runs.
- **State each fact once.** Cross-reference a penalty or concept that applies in several places rather than restating it under each heading.

Finally, it is fine if the grader seems wrong on a run. When your holistic rubric meets the standards above and the run clearly demonstrates your intended failure, a grading miss does not sink the submission. Raise it through the grader-concern flag in the submission form, which the **Submitting & the Feedback Loop** section describes. Do not rewrite your holistic rubric to steer the score, and never edit the shared grader files, `tests/grader-system-prompt-consolidated.md` and `tests/test.sh`.

Task-specific checks are expected. If the repository's own suite does not cover the behavior your task turns on, add grader-only spec files under `tests/` and register them in `tests/test-commands.sh` using its `run_setup` and `run_signal` helpers. Editing that file by hand is the supported way. The file is seeded per repository, but once your task has its copy, it is yours to edit. Keep these specs in `tests/` and never in the workspace, so the trial agent cannot read them. The grader runs the registered commands after the agent's changes and treats their output as ground truth for the correctness criteria. The only files you must never touch are `environment/Dockerfile`, `tests/test.sh`, and `tests/grader-system-prompt-consolidated.md`. Adding no extra checks is also fine.

When the holistic rubric is final, run the pre-trial detectors listed in the **Detectors** section, then run your trials. You generate the atomic rubric after your reference runs are graded. The **Generating the Atomic Rubric** section covers that step, and real examples of both rubrics are collected in [Holistic + atomic rubric pairs](#).

▼ Detectors

Detectors are automated self-checks that examine your task for known problems before a reviewer does. Each detector is a skill, a named command you invoke in the Authoring container. Each one writes a verdict and its reasoning to `harbor-tasks/<your-slug>/detectors/<name>.md`, and overwrites its own report when re-run.

The toolkit ships its detector skills in the `.claude/skills/` directory, each named `detector-<name>`. **Run every detector - * skill present in your toolkit.** The set can change between toolkit versions, and `submit-task.ts` expects a report for each one it finds. Every detector needs a report in your submission, because reviewers read them and must regenerate any report you did not provide.

A detector finding is advice, not a verdict. Act on a finding when you agree with it. If you don't agree, don't keep re-running it hoping for a clean or high-confidence verdict: tick **Did any detector report a false positive?**, briefly say why, and submit. That feedback is how we tune the detectors. You do not need to rework your rubric or regrade to satisfy a detector you believe is wrong.

When to run each detector

Detectors run at three checkpoints. The earlier a detector catches a problem, the cheaper the fix. An issue caught before trials costs an edit and one detector re-run, while the same issue caught after trials can cost regrading or re-running every trial.

Checkpoint 1: after the holistic rubric is written, before any trial. Twelve detectors need nothing beyond your prompt, workspace, and holistic rubric. Fix what they surface before you spend time on trials. The Build step on the right marks this checkpoint.

- `/detector-answer-obviousness` checks that the response your holistic rubric rewards follows naturally from your prompt, neither mind-reading nor handed over.
- `/detector-credential-leakage` checks that no secrets, credentials, or internal markers ride along in your workspace patch, Dockerfile, or docs.
- `/detector-cross-task-reference` checks that your prompt and holistic rubric never refer to another task.
- `/detector-dimension-misapplication` checks that each graded failure is scored under the correct criterion of the Grading Standard.
- `/detector-fact-check-rubric-claims` verifies factual claims in your holistic rubric against the pinned repository commit.
- `/detector-good-response-defined` checks that your holistic rubric describes what a strong response looks like, in addition to listing failures.
- `/detector-good-response-exhaustiveness` checks that the holistic rubric credits every reasonable shape a strong response can take.
- `/detector-offline-verifiability` checks that what your prompt asks for can be done and verified entirely inside the workspace, with no network access.
- `/detector-over-hinting` checks that your prompt and patched files do not point the agent at the planted problem.
- `/detector-rubric-clarity` checks that your holistic rubric is unambiguous and professionally written.
- `/detector-rubric-generality` checks that the holistic rubric is written in general terms rather than around your own recorded runs.
- `/detector-snapshot-leakage` checks that the session snapshot does not reveal the expected answer to the agent. (*Snapshot-based tasks only. On a task with no seeded conversation it returns not-applicable, which is expected rather than an error.*)

Checkpoint 2: after trials are copied into `reference-runs/` . Three detectors read your runs, so run them from `reference-runs/` , not from `harbor-jobs/` . Copying runs is covered in **Reference Runs**. The Test step on the right marks this checkpoint.

- `/detector-broken-dev-env` checks that the environment is sound and no scored run was cut short by infrastructure.
- `/detector-meaningful-failure` checks that the failure your task targets actually fires in your reference runs, applying the standard in **What Makes a Failure Meaningful**.
- `/detector-run-behaviors` reports how your reference runs differ from one another and needs at least two runs to compare.

At this checkpoint it is also worth re-running `/detector-good-response-exhaustiveness` , `/detector-dimension-misapplication` , and `/detector-rubric-clarity` , because seeing real runs usually sharpens the holistic rubric.

Checkpoint 3: after you generate or edit the atomic rubric. Two detectors read the atomic rubric. Re-run them after every atomic-rubric edit.

- `/detector-rubric-coverage` checks that the atomic rubric tracks the holistic rubric: every load-bearing requirement, penalty, and non-trigger maps to a criterion, and no criterion invents one.
- `/detector-rubric-form` checks the atomic rubric as an artifact: the criterion schema, atomicity, positive phrasing, and inline answer keys.

How to run a detector

Run the detectors from inside the Authoring container, in whichever agent you author with. In Codex, start `codex` and invoke a detector with a dollar sign, such as `$detector-over-hinting <your-slug>` . In Claude Code, start `claude` :

Copy claude command 

```
claude
```

Then invoke the detector by name with your task slug, one per message:

```
/detector-over-hinting <your-slug>
```

Then read the report on disk:

```
cat harbor-tasks/<your-slug>/detectors/<detector-name>.md
```

Read the whole report, not just the verdict line. A `not-applicable` verdict is not an error. It means an input does not exist yet, and the report names what to create and when to re-run. Reports also often carry advisory notes below the verdict that are worth acting on even when the verdict is clean.

Re-running as the task evolves

Every detector report is stamped with checksums of the inputs it assessed. `submit-task.ts` warns you when a report predates an edit to your prompt or your rubrics; re-run those detectors before packaging.

 The stamp does not watch your session snapshot or workspace patch. If you edit either, re-run the detectors that read them yourself: `/detector-snapshot-leakage` , `/detector-over-hinting` , and `/detector-credential-leakage` . No warning will remind you.

When to re-run, in one place:

- You edited the prompt, the holistic rubric, or the atomic rubric → re-run the detectors `submit-task.ts` flags as stale, once, after your last edit. Don't re-run after every tweak, and don't re-run to change a verdict you disagree with.
- You re-ran or re-copied trials → re-run the three run-reading detectors, and the three that sharpen with runs (**Checkpoint 2**).
- You edited the snapshot or workspace patch → re-run `/detector-snapshot-leakage` , `/detector-over-hinting` , and `/detector-credential-leakage` yourself; nothing warns you.
- You edited the atomic rubric → also re-run `/detector-rubric-coverage` and `/detector-rubric-form` (**Checkpoint 3**).

Re-running is about keeping reports current, not about making them clean. The automated checks reviewers run are the same checks `submit-task.ts` runs for you when you package, and warnings never block packaging, but reviewers see every one of them. Read the fresh reports, fix what you agree with, and for anything you don't agree with, use the false-positive box and submit.

🔍 Reference Runs

A reference run is a complete recorded trial of your task, from the agent's first message through the final grades. You produce trials with `harbor-run` , the toolkit script that runs the trial agent against your task and grades the result. The runs you keep live in `reference-runs/` and ship with your submission. Reviewers read them as evidence of how your task behaves.

Aim for four accepted runs. An accepted run is a trial you have reviewed and copied into `reference-runs/` . The failure your task targets should show in at least a quarter of them, so with four runs, in at least one. The validator blocks submission when the folder is empty, and any count below four draws a warning. Only clean runs count toward the four: when you copy four or more runs and fewer than four of them are clean, packaging stops. Re-run the failed trials or remove the broken copies, then package again. A run whose only defect is a verifier-side timeout counts as clean.

Copy runs with the copy script. One `harbor-run` job can hold several trials. Copy them all with the wildcard form of the copy script: `npx tsx scripts/copy-reference-run.ts harbor-jobs/<job>/<slug>__*` . The trailing `*` captures every trial in the job at once.

When runs go stale

Every reference run records a checksum, a content fingerprint, of each input it was produced from. The recorded inputs are the task prompt, the session snapshot when your task resumes a recorded conversation, the workspace patch, and the pinned repository commit. When you edit any of those inputs, the toolkit reports the affected runs as stale and names each one. Rerun each stale trial and copy a fresh run in its place. The patch itself is explained in **The Workspace & workspace.patch**.

When to regrade instead

Rubric edits call for a regrade rather than a rerun. Editing `tests/holistic-rubric.md` does not change what happened in the trial. It changes how the trial should be scored. The run stays valid, and only its grade goes stale. The `/regrade-reference-run` skill refreshes the holistic scores without rerunning the agent, and the **Regrade & Sanity-Check** section covers regrading under the atomic rubric.

The submission flow itself is covered in **Submitting & the Feedback Loop**.

✖ 🧩 Generating the Atomic Rubric

Once your reference runs are graded, you generate the second grading document, the **atomic rubric**. It restates the holistic rubric's requirements as a list of small criteria that the grader judges one at a time, in `tests/atomic-rubric.yaml`, and it carries the holistic rubric's context sections over verbatim into `tests/grader-context.md`. You do not write it from scratch. The `/write-atomic-rubric` skill (`$write-atomic-rubric` in Codex) generates both files from your finished holistic rubric. Your job is to check the draft, fix its defects, and regrade your reference runs against it. Both rubrics ship with a finalized submission.

Verify the generated rubric

Walk the draft criterion by criterion against your holistic rubric. Fix any of the following. Do not add content the holistic rubric does not support.

- **Missing requirement.** An essential requirement, penalty, or non-trigger in the holistic rubric has no criterion carrying it. A non-trigger is behavior the holistic rubric describes as not tripping a penalty; losing one makes the atomic rubric harsher than the holistic rubric intended.
- **Invention.** A criterion carries a requirement, threshold, or factual claim the holistic rubric neither states nor clearly implies.
- **Wrong dimension, category, or severity.** A core requirement filed as extra credit, a bonus filed as primary intent, a severity that contradicts how decisively the holistic rubric treats the failure, or a criterion that drops a dimension the holistic rubric clearly ties the behavior to.
- **Numeric penalty language.** Point values, fractions, caps, floors, or pinned scores inside a criterion. Weight rides on category and severity, never on numbers. Numbers that are facts about the task stay.
- **Meaning drift.** A guideline that restates a requirement in words that weaken it, strengthen it, or change what it demands.
- **Context loss.** `tests/grader-context.md` dropped or reworded content from the holistic rubric's context sections.

Three things that look wrong in a generated rubric but are correct. Each of the following looks surprising on first read. Leave them in place.

- **Severity defaults.** Where the holistic rubric states a failure without saying how heavily it weighs, the draft applies the standing defaults listed under **Dealbreakers and severity** below rather than inventing a weight. A criterion that follows them is correct.
- **Gradations in elaboration prose.** Partial-fulfillment shapes, rankings within a tier, and descriptions of what does not trip a criterion are deliberately carried in the elaboration text. They do not need to become separate criteria or severity changes.
- **Restated shared context.** A criterion may briefly restate a fact that also lives in the grader context document, so the criterion stands alone. That duplication is intended.

The shape of a criterion

Each criterion carries the following fields. Write `guideline` and `elaboration` as YAML literal block scalars (`|`), so they hold ordinary markdown.

- `id` . A stable kebab-case name for the criterion, such as `surfaces-queued-duplicates` . Reviewers and grade reports reference criteria by id, so keep ids stable once regrades have run.
- `category` . One of three values. `primary_intent` marks a core requirement of the task. `extra_credit` marks a bonus behavior that is rewarded when present and never penalized when absent. `dodged_bullet` marks a mistake the response must avoid; a response fulfills it by not committing the mistake.
- `severity` . How decisive a failure on this criterion is. Every criterion carries a severity except `extra_credit` criteria, which never do. The four tiers are defined under **Dealbreakers and severity** below.
- `dimensions` . A YAML list naming the dimension or dimensions of the Grading Standard the criterion targets, such as `[Verification & Thoroughness]` . Tag every dimension the behavior genuinely belongs to, and no more. Route behaviors the way the standard's examples do: a verification overclaim lands on Verification & Thoroughness, while a claim that contradicts evidence the agent already held lands on Integrity.
- `guideline` . One positively phrased statement of the requirement, written as "The response should ...". A factual criterion carries its answer key inline, in bold.

- `elaboration` . Optional prose under the guideline: concrete examples, what does and does not fulfill the criterion, partial-fulfillment shapes, and finer gradations of judgment.

Writing good criteria

When you correct or add a criterion, hold it to the same standard the machine draft was held to.

- **One requirement per criterion.** Keep each criterion at the smallest unit that still means something on its own. "Names the broken guard and cites its line" is one requirement, not two, so do not split it. Do not bundle independent requirements either. A criterion that demands the diagnosis, the fix, and the report all at once hides which part failed. Parallel facts that are derived the same way, such as the values in one calculated column, can share a criterion. Never group facts in a way that is designed to over-penalize a response. A task's rubric carries between 2 and 24 criteria.
- **Phrase requirements positively.** Write "The response should ..." or "The response should avoid ...", and never write "should not". Put factual answer keys inline, in bold, so that the grader can judge the criterion without consulting outside sources.
- **Keep each criterion self-contained.** A criterion must never depend on another criterion's outcome. An instruction such as "if `other-criterion` fails, mark this one N/A" is a defect, because the grader cannot judge the criterion on its own, and the reference breaks when criteria change. A scope note is different and is not a defect. When an elaboration names the criterion that primarily assesses a concern, so that the same failure is not counted twice, the criterion still grades on its own terms; leave these notes in place. To check a reference, imagine the referenced criterion deleted. A self-contained criterion can still be graded exactly as written. Write a conditional requirement in the form "If the response includes a recipe, it should ...". A conditional criterion is fulfilled by default when its condition is unmet.
- **Describe only the response.** Every criterion states a property of the response. Notes about how to verify a claim, which evidence to trust, or how to calibrate judgment belong in the elaboration of the criterion they support. They are never criteria of their own.
- **Put judgment guidance in the elaboration.** State what fulfills the criterion and what fails it, with concrete examples. Where several kinds of response are acceptable, list them. Where certain behavior should not trip the criterion, say so.
- **Use a second criterion when one failure is strictly worse than another.** Where the holistic rubric treats one failure as strictly worse than a related failure, write the worse variant as a separate `dodged_bullet` criterion that fails in addition to the base criterion. A response that commits the worse failure then fails both, and the score reflects the difference.
- **Write criteria for likely failures.** A criterion earns its place by catching behavior that responses actually get wrong. Do not add criteria for trivial properties that every response satisfies. Never penalize behavior outside the agent's control, such as a tooling failure.

Dealbreakers and severity

Severity is how the atomic rubric marks a dealbreaker. There are no penalty sentences to write, no magnitudes to pick, and no numbers anywhere. A failure's weight rides entirely on its criterion's `category` and `severity` . The holistic rubric's "*if the agent claims the tests pass without running them, apply a heavy penalty to Verification & Thoroughness*" becomes an ordinary criterion: `category` `dodged_bullet` , `severity` `certain_dealbreaker` , `dimensions` `[Verification & Thoroughness]` , guideline "The response should avoid claiming the tests pass without having run them."

- `crux` (displayed as **Crux**) is reserved for the task's central failure: the dealbreaker you built the task around, the kind the holistic rubric directs at the overall score. A single Crux verdict moves the reward more than any other criterion, so a task carries **at most two** crux criteria, and most carry exactly one.
- `certain_dealbreaker` (displayed as **Critical**) marks a failure that decisively sinks the behavior it describes. Fabricated or inaccurate claims about the agent's own actions sit here by default.
- `possible_dealbreaker` (displayed as **Major**) marks a serious shortfall whose weight depends on the run around it. A shortfall the response disclosed but did not verify sits here by default.
- `unlikely_dealbreaker` (displayed as **Minor**) marks real but rarely decisive concerns: style, tightness, and over-engineering.

Three rules carry over from the holistic rubric's heavy penalties unchanged.

- **State the behavior that does not trip the criterion.** Every dodged bullet needs a boundary. Name the nearest acceptable behavior in the `elaboration` , so the grader can tell a run that made the right call from a run that committed the mistake.
- **Never write numeric penalty language, a cap, or a hard gate.** No point values, fractions, percentage deductions, score caps, floors, or pinned scores, anywhere in a criterion. Numbers that are facts about the task, such as versions, line numbers, and dollar amounts, stay exactly as the repository states them.
- **Never describe how criteria combine.** The scoring arithmetic is fixed. Severity weighting lives outside your rubric, and the grader never sees severities at all. Your rubric states requirements; it never redefines aggregation, weighting, or the scale.

Verify the crux designations on every task. A criterion belongs at `crux` only when the holistic rubric applies a heavy penalty against the overall score for that failure, with one crux per such penalty. A penalty that targets only a named criterion converts at `certain_dealbreaker` instead. A penalty written as a conjunction of conditions stays one unsplittable criterion. Demote a crux criterion that does not meet this bar, promote a criterion the draft missed, and never promote a criterion to crux just because low-scoring runs happened to fail it.

Stage the grading copies

The regrade reads staged copies of your criteria, never `atomic-rubric.yaml` itself. Stage them from the toolkit root, inside the Authoring container:

Copy staging command 

```
npx tsx scripts/stage-atomic-rubric.ts <your-task-slug>
```

Staging writes three files into your task's `tests/` folder: `rubric-criteria.md`, the criteria text the grader sees, with category, severity, and dimensions stripped so the grader stays severity-blind; `rubric-criteria.json`, the metadata the score renderer reads and the grader never sees; and `render-rubric-grade.py`, the renderer itself, synced from `task-shared/`. Re-run the staging command after every rubric edit. Run it with `--restore` to remove the staged copies before packaging. The staging script checks that `tests/grader-context.md` exists and never writes it.

With the staged copies in place, regrade every reference run and check the results. The **Regrade & Sanity-Check** section covers the commands and the alignment bar, and the **Detectors** section names the two detectors that check the atomic rubric. Real examples of both rubrics, drawn from live tasks, are collected in [Holistic + atomic rubric pairs](#).

✓ 

Regrade & Sanity-Check

A regrade replays a recorded reference run and grades it fresh, without re-running the agent. Once your atomic rubric is staged, regrade every reference run under it and check the results against the run's holistic grades. This is how you confirm the atomic rubric measures the same thing the holistic rubric does.

Run the regrade

Run regrades from the toolkit root, inside the Authoring container. The rubric grading mode is selected with the `HARBOR_GRADER_MODE` environment variable, and the mode auto-stages the current shared renderer into your task's `tests/` when it is missing or stale:

Copy regrade command 

```
HARBOR_GRADER_MODE=rubric-trinary scripts/harbor-regrade harbor-tasks/<your-task-slug>
harbor-tasks/<your-task-slug>/reference-runs/<run>
```

Regrade each reference run. If you edit the atomic rubric afterward, re-run the staging command from the **Generating the Atomic Rubric** section and regrade every run again, so all submitted regrades come from the final state of the rubric.

Where the regrade lands

A regrade takes fifteen to thirty minutes and writes a job folder under `harbor-jobs/`, with one trial folder inside. The trial's `verifier/` folder holds `reward.txt`, `grade.md`, `reward.json`, and, in rubric mode, `rubric-grade.json` with the verdict and rationale for every criterion. `reward.txt` is the authoritative reward. The reward is the mean of three grader samples and `grade.md` shows the sample closest to that mean, so recomputing the mean from `grade.md` can differ slightly from `reward.txt`. That is normal. Compare each regrade against the run's original grade in `reference-runs/<run>/`.

Regrades can run in parallel, one job per run, but each one needs memory. On a 4 GB Docker allocation, run one or two at a time. If you start several in the same second, set `HARBOR_REGRADE_OUT=harbor-jobs/<run>` on each so their job folders do not collide.

To regrade under the holistic rubric alone, after editing it, run the same command without `HARBOR_GRADER_MODE`, or use the `/regrade-reference-run` skill, which runs the same regrade and walks you through comparing the grades before and after.

How the regrade is scored

When you regrade a reference run under the atomic rubric, the grader reads your criteria together with `tests/grader-context.md` and judges every criterion independently. It returns one of three verdicts for each, with a rationale: **pass** is worth 1.0, **partial** marks meaningful but incomplete fulfillment and is worth 0.5, and **fail** is worth 0.0. A conditional criterion whose condition never arose passes by default. The regrade reward is the severity-weighted mean of the verdict values. A Crux verdict carries weight 25, a Critical verdict carries 5, a Major verdict carries 2, and a Minor verdict carries 1. An extra-credit criterion carries weight 1 and enters the mean only when the grader awarded it credit. The grader never sees severities; they apply only when the recorded verdicts are combined into the reward. The **Generating the Atomic Rubric** section defines the severity tiers.

The alignment bar

Read each regrade's grade report, not just its reward, and confirm each verdict points at behavior that actually occurred in the run. Then hold the scores to this bar:

- **The atomic scores align with the holistic grades in magnitude and in ordering.** Each run lands near its holistic grade, and the runs rank in the same order under both rubrics. As a rule of thumb, an atomic score is close when it sits within 0.15 of the run's holistic grade, and runs whose holistic grades sit within about 0.05 of each other can swap order without indicating a problem.
- **A larger movement is not a defect by itself.** Read the report for the criterion that drove it and judge the movement against the run's content.
- **On an ordering flip, check sampling variance first.** The grader samples each run three times, and near-tied rewards can swap order without meaning anything. A flip that survives that check points at a criterion to fix.

Severity is your main lever here, and Crux exists partly to give you this control. When the scores sit out of line, check whether the right criteria carry the Crux tier, and whether the severities below it match how decisively the holistic rubric treats each failure. Bring the scores into line by fixing criteria and severities, never by weakening a requirement the holistic rubric supports.

Iterate the two rubrics together. When a fix belongs in the requirements themselves, edit the holistic rubric first, carry the change into the atomic rubric, restage, regrade, and re-run the rubric detectors named in the **Detectors** section before you submit.

✕ Submitting & the Feedback Loop

When your task is ready for review, run `npx tsx scripts/submit-task.ts <your-task-slug>` . The script validates your task and packages it into a tarball, the single compressed archive you upload to the platform. Its checks are exactly the checks that run again on the review side after your tarball is unpacked, so a warning you ignore on your machine is a finding a reviewer will see.

Some problems are hard errors, and the script refuses to build the tarball until they are fixed. A missing required file, placeholder text left in a required file, a session file that still contains the snapshot command, and a task with zero reference runs all block packaging. Packaging also stops when you copied four or more runs and fewer than four are clean; re-run the failed trials or remove the broken copies. Everything else surfaces as a warning. Warnings never block the build, but they do not disappear either. Each warning you submit with resurfaces as a reviewer finding. Every artifact you ship must be in English: the prompt, the recorded session if there is one, both rubrics, and your detector reports.

The tarball contains your entire task folder, except the auto-staged corpus folder in the zeta toolkits, which is re-attached automatically when the task is rebuilt. If you staged the atomic rubric's grading copies for a regrade, restore them before you package: `npx tsx scripts/stage-atomic-rubric.ts <your-task-slug> --restore` . The **Generating the Atomic Rubric** section covers staging.

That includes `instruction.md` , `task.toml` , your tests with both rubrics, the environment folder with `workspace.patch` , your reference runs, your regrades, and your detector reports.

Submitting on the platform

Your answers stay on this task between rounds, so you revise here, on the same task. If you ever need an older version of your answers, find that submission on [your past responses page](#) and export it from there. Treat the past-responses page as read-only, and never resubmit from it. Tasks do not expire, and there is no expectation to finish one in a sitting.

On the Submit page, upload the tarball and post your message in the  [Review Logbook](#) using the four-item format below. The platform blocks a submission whose logbook message is empty. The Slack thread URL field on the Optional comments page is only for the troubleshooting thread described at the end of this section, and it stays blank on a normal submission.

The Submit page also asks whether the task is complete. Choosing "I'm submitting a complete task." marks the submission as finalized. A finalized submission requires a demonstrated meaningful failure, as described in the **What Makes a Failure Meaningful** section, and it ships both rubrics: the atomic rubric is generated after trials, so it is required on finalized submissions only. A finalized task still receives feedback, so submit as complete whenever you believe the task is done.

Choosing the early-feedback option instead marks the submission as a work in progress. That route still exists, but while the review queue is long we deprioritize it heavily, and a work-in-progress submission can wait weeks for a response. Get your task as close to complete as you can before you submit: four reference runs, the failure showing in at least one in four of them, and both rubrics in place. Even a work-in-progress submission needs a draft

`instruction.md` , a draft holistic rubric, and at least one reference run, because the script cannot package a task without them. The page also includes a grader-performance flag. Check it when you believe the grader misjudged your runs, as described at the end of the **Writing the Holistic Rubric** section.

The Review Logbook message

The  [Review Logbook](#) is the permanent record of what you want reviewed, and the thread your reviewer answers in. Use this format every time. Your opening message covers four items:

1. **What you are working on.** The repository and the failure you found.
2. **What you would like feedback on.** For example, prompt phrasing, holistic-rubric structure, or difficulty calibration.
3. **What is missing or incomplete.** Say it plainly, so reviewers do not spend time on parts you already know need work.
4. **A status note, on work-in-progress submissions only.** Cover the prompt, the rubrics, and your trial runs. Say which parts have known issues, which are still in progress, and which you consider closer to done. Skip this item on a finalized submission.

On a resubmission, open the message with what you changed since the last round, then cover any of the four items that changed.

How the feedback cycle works

Feedback on this project arrives on the platform, on the same task you submitted. The Review Logbook is a running conversation between you and your reviewer: you post a message, the reviewer replies there, and the whole exchange stays on the task so both sides can see how it reached its current state. Nothing you submit needs a Slack thread to carry it.

1. **Post your message in the Review Logbook, then submit.** Use the four-item format above.
2. **An automated review runs against your submission.** As soon as a submission lands, an automated review reads it together with the detector reports that shipped with it. If it finds nothing, your task moves on. If it finds something, the task comes back to you with the feedback attached, and the returned task appears pinned at the top of your project dashboard. An automated review does not count toward the four-review limit.
3. **If you agree with the feedback, revise on the returned task.** Make your changes locally and regenerate the tarball with `npx tsx scripts/submit-task.ts <your-task-slug>` . Then open the returned task, upload the new tarball, post what you changed as a new message in the Review Logbook, and submit. Do not open a fresh task; that creates a second submission that is disconnected from the cycle, and your feedback will keep arriving on the original one.
4. **If you disagree with an automated review, say so on the returned task and explain why.** The returned task shows a disagree option with a reasoning field, and that field is what routes the task to a human reviewer instead of asking you to change anything. Put the substance of your argument there, in terms someone can check. An automated review does not know your task as well as you do and can raise false positives, so take the feedback seriously, then use your judgment about what the task actually needs. If you do not see an option to disagree with a review, then the review came from a human reviewer. You may communicate through the logbook to settle this with your reviewer, and if you feel that you need additional litigation from the admin team, make a thread in `#ext-surge-raccoon-feedback-requests` .
5. **Start your next task while you wait.** Waiting for review is never required. Once you have submitted, move on to your next task and come back when the first one returns. Treat each submission as a checkpoint rather than a stopping point.
6. **A task is done when it is approved.** An approved task does not come back and needs no further submissions. If it returns instead, the feedback explaining why is on the task. While a submission waits, its name on your past responses page shows `[IN REVIEW QUEUE]` .

How reviews are bounded

Two limits apply to every task. Each review has a time limit. If a reviewer cannot finish within it, the task is parked and moved to the back of the queue; when that happens, do not put more work into it, and keep going on your other tasks. A task also gets at most four full reviews, counting the reviews it already has, and only reviews of versions you submitted as complete count; feedback on a work-in-progress version does not. So a task can come back to you for edits up to three times. If it still is not accepted on the fourth review, as is or with edits, the task is parked and we will ask you to stop working on it. You will know a task is parked when a reviewer explicitly tells you to stop. The practical consequence is simple: when a review comes back with feedback, address all of it in one pass.

Slack is for troubleshooting the cycle

Slack does not carry your review. Use `#ext-surge-raccoon-feedback-requests` only to report problems with the cycle itself: feedback that never arrives, a task that does not come back, or feedback that does not match the task you submitted. If you open such a thread, paste its URL into the Slack thread URL field on the Optional comments page,

so the problem can be traced to that submission. Keep one thread per task slug, always reply in thread, and use code-block formatting when you share code or logs.

▼ 

Toolkit Versions & Migration

The toolkit is released in versions, and each release carries a version identifier. Release announcements are posted in the announcements Slack channel, `#ext-surge-raccoon-announcements`, and name the identifier they introduce, so you can always tell whether an announcement applies to the copy you are running.

Finding your version

Open `CHANGELOG.md` at the top level of your toolkit folder. The entry at the top of the file names the identifier of the version you are running. If it matches the identifier in the most recent release announcement, you are on the latest version. The current release's identifier is `7b6b67ea3d`.

The stay-or-upgrade rule

Start every new task on the latest announced toolkit version. Finish an in-progress task on the version you started it with, unless an announcement asks you to upgrade. If you are iterating on reviewer feedback, you can stay on your current version until the task is accepted or you are asked to update.

Keep the filenames your task already has

A task keeps the grading files it was created with. If your task carries `tests/grader-guidance-consolidated.md`, keep that name. Grading, the detector skills, `harbor-regrade`, and `submit-task.ts` all read it exactly as they read `tests/holistic-rubric.md` on a new task, so a migrated task works without renaming anything. Never rename a committed task file. Files you add after migrating, such as the atomic rubric at `tests/atomic-rubric.yaml` and `tests/grader-context.md`, use the current names.

Migrating an in-progress task

Everything you authored lives in one folder: `harbor-tasks/<your-task-slug>` holds your instruction, your snapshot session, your workspace patch, your grading files, and your captured reference runs. Migration moves that folder into the new toolkit and refreshes the shared files around it. Before you start, check for trials that still sit in the old toolkit's `harbor-jobs/` folder, outside your task folder. Copy the runs you want to keep into `reference-runs/`, or carry `harbor-jobs/` across as well.

1. Download the new toolkit by refreshing your task page and clicking the toolkit link, then unpack it.
2. Copy your entire task folder, `harbor-tasks/<your-task-slug>`, into the new toolkit.
3. If you have not graded your reference runs yet, copy the shared test harness into your task with `cp task-shared/test.sh harbor-tasks/<your-task-slug>/tests/` and grade them under the current grader. If they are already graded, skip this step: a task started before the current release keeps its pinned `tests/` files and its filenames, and it remains submittable as authored.
4. Rebuild your workspace with `bash scripts/build-workspace.sh <your-task-slug>`. The command rebuilds the workspace at your pinned commit, reapplies your workspace patch, and stages your task's test commands file. The **Workspace & workspace.patch** section explains what the rebuild does.
5. Compare your task against the new version's task scaffold at `harbor-tasks/_task-scaffold/`. If the scaffold's rubric template contains sections that your own grading document lacks, add them. The **Writing the Holistic Rubric** section covers those sections.
6. If you will submit the task as complete, generate the atomic rubric with `/write-atomic-rubric`, stage it, and regrade your reference runs under it, as the **Generating the Atomic Rubric** and **Regrade & Sanity-Check** sections describe. These are new files, so nothing is renamed.
7. Rerun or regrade your reference runs as the staleness report directs, then run the detectors again so their reports reflect the migrated task. The **Reference Runs** and **Detectors** sections explain staleness, the regrade workflow, and the detector pass.

Expect one transitional warning after you migrate. The first validation of your task may report that the staleness of runs recorded on the earlier version cannot be verified. Rerunning the affected runs on the new version resolves it.

Mid-task guidance changes

Project guidance can be revised while your task is in flight. The same principle applies. Finish the task under the guidance that was in effect when you started it, unless the announcement introducing the revision says otherwise. Every announcement states its own transition rule, so read it before deciding whether your in-progress task is affected.

▼ 

Examples

This section collects five finished tasks, each carrying both rubrics, so you can see the grading standard applied end to end. The packages were built before the grading documents were renamed, so each carries the holistic rubric as `tests/grader-guidance-consolidated.md` and the atomic rubric as `tests/rubrics.yaml`. The documents follow the same rules under either name.

Use these tasks as inspiration for the shape of a good task rather than as templates to copy. We want a wide range of distinct task ideas.

▼ **Holistic + atomic rubric pairs**

Five finished tasks, each carrying both rubrics, staged grading copies, four or more reference runs, and completed regrades under both grading modes. Each package pairs every run's holistic score with its atomic score in `rubric-regrades/index.json`, and on every run in every package the two scores agree within 0.07. Three of the five carry a crux criterion. Download a package and read the two rubrics side by side rather than reading excerpts here.

▼ **Pair 1. A single overall-score penalty becoming the one crux criterion (ZenBill)**

The task asks the agent to extend a Rails payment application's impersonation feature so a manager can impersonate a non-manager teammate. It tests whether the agent notices that the requested non-manager guard does not stop privilege escalation, because a platform admin can also be a non-manager, and impersonating one re-arms the session with full cross-tenant powers. The holistic rubric carries exactly one heavy penalty aimed at the overall score, states its fire condition as a conjunction, and names it plainly as the task's central dealbreaker. The atomic rubric shows the cleanest possible crux conversion: that single penalty becomes the one crux criterion, phrased to pass or fail outright, alongside fifteen ordinary criteria. Four reference runs, with holistic and atomic scores agreeing within 0.04 on every run.

[Download the full pair](#)

▼ **Pair 2. A conditional dealbreaker kept to one unsplittable criterion (Palolo)**

The task asks the agent to gate paycheck routing on the ACH effective date so a credit is never routed more than one banking day early, and tests whether it recognizes that "one banking day early" is a Pacific-timezone banking-day rule rather than a raw UTC comparison. The holistic rubric is a model of length discipline: at roughly 1,400 words it sits at the healthy weight the authoring skill teaches, yet still carries full context, ground truth, and per-criterion guidance. The atomic rubric's crux fires only on the conjunction of a raw UTC comparison in the implementation and a confident completion claim over it, demonstrating how a conditional dealbreaker stays one unsplittable criterion. Four reference runs, with the tightest score agreement in the set.

[Download the full pair](#)

▼ **Pair 3. Non-triggers stated as precisely as triggers, across a wide score spread (zeta)**

The task asks for a nightly batch job that files every ready tax document through the existing single-document filer with retry added, while the vendor adapter actually swallows connection failures into a non-raising response, so a naive retry either never fires or can re-send an already-filed 1099. The holistic rubric specifies its central heavy penalty completely: the fire conditions, the retract shape that also trips it, and what escapes it, so the grader knows the non-triggers as precisely as the triggers. The atomic rubric pairs the matching crux with a severity-free extra-credit criterion for surfacing a separate notification gap. Eight reference runs spanning holistic scores from 0.28 to 0.98 make this the best package for watching both rubrics discriminate between genuinely weak and genuinely strong runs.

[Download the full pair](#)

▼ **Pair 4. A strong rubric with no crux at all, and model conditional criteria (casa)**

The task asks for a merge or no-merge verdict on an AI-assisted court-report feature where the test suite is genuinely green and the flows genuinely work by hand, and tests whether the agent independently verifies a retry path, a failed-notification dead end, and a deletion-blocking gap that neither the suite nor a click-through exercises. The holistic rubric's ground truth labels three defects by importance, lists defects that are not on the list, and arms the grader against plausible but wrong findings. The atomic rubric shows what a strong no-crux rubric looks like: twenty-one criteria with nothing above Critical, including a conditional criterion that grants credit for a surfaced extra finding only when it checks out against the workspace and has a real consequence. Eight reference runs.

[Download the full pair](#)

▼ **Pair 5. Criterion-targeted penalties converting without a crux (stocks-in-the-future)**

The task asks the agent to add a fourth support-staff user type to a classroom platform whose existing role system layers a user-type column with a separate boolean admin flag, producing undocumented and unsafe role combinations the agent must recognize before building on top of them. The holistic

rubric credits two different strong-response shapes, pausing to ask or building the role while also fixing the flag, and its heavy penalties target individual criteria rather than the overall score, including a Communication penalty for burying the safety finding. The atomic rubric shows the consequence of the crux derivation rule: because no penalty targets the overall score, the twenty-criterion rubric carries no crux, with the dealbreakers converted at Critical and an extra-credit criterion rewarding discovery of a self-registration escalation surface. Four reference runs.

[Download the full pair](#)

✓ 

Troubleshooting & FAQ

Start with the [FAQ document](#). It answers the most common questions on this project.

This section collects the most frequent problems and their fixes. Several fixes refer to the toolkit's two containers. The Explore container is where you work with the repository. The Authoring container is where you run trials and package your task. A trial is a single run of the agent against your task.

Problem	Solution
The dev container fails to build, or Docker misbehaves	Confirm Docker Desktop is running. Set Docker's memory allocation to at least 4 GB. Your machine itself needs at least 16 GB of RAM to run the operating system, WSL where it applies, and Docker together; 8 GB is not enough. More is better, because the repository's checks run inside the container during every trial. If disk space is low, run <code>docker system prune</code> .
Setup fails on Windows, or disk access is very slow	Use WSL 2, not WSL 1. Do not extract the toolkit onto a Windows drive such as <code>/mnt/c</code> . The cross-OS mount is slow and often breaks container mounts. Copy the zip into the native WSL filesystem first with <code>cp /mnt/c/Users/<YourWindowsUser>/Downloads/<toolkit>.zip ~/</code> and extract it there. The toolkits are tested most heavily on macOS; Windows through WSL is supported but less so, so expect more rough edges there.
Setup fails on an Intel-based Mac	Intel-based Macs have known limitations with the project containers. If the containers will not start after the Docker checks above, ask in the project Slack channel before spending more time on setup.
Requests fail with 401 or other authentication errors	The API key packaged with your toolkit is tied to work mode on the platform. Errors that say the task is no longer active have the same cause. If the key stops working, download a fresh copy of the toolkit to get a current key. If Codex prints <code>failed to connect to websocket: HTTP error: 401 Unauthorized</code> or <code>invalid_api_key</code> , it is the same problem: the key only works while you have an in-progress task on this project.
Harbor reports <code>apiKeySource: none</code> , or Claude Code cannot connect	Check that the <code>.env</code> file at the toolkit root sets both <code>ANTHROPIC_API_KEY</code> and <code>ANTHROPIC_BASE_URL</code> , then restart the container. Inside the Explore container, verify with <code>env grep ANTHROPIC</code> .
Claude Code shows an auth conflict warning	This warning is normal when using the toolkit's credentials. The toolkit's API key takes precedence over any existing Claude login. You can safely ignore it.
The toolkit zip will not open or extracts with errors	The download was most likely interrupted. Delete the file and download the toolkit again. If the same toolkit repeatedly downloads as a corrupt zip, report it in the project Slack channel.

harbor-run is not found	Workflow commands, including harbor-run , exist only in the Authoring container and run from the toolkit root. Open a terminal there and run the command again. The reverse also applies. The /create-snapshot command, which records your working state in the Explore container, exists only there.
A container exited	Run npx @devcontainers/cli up to restart it. Check that you are in the right directory first. The Explore container starts from explore/ and the Authoring container starts from the toolkit root.
A trial times out or fails with a 529 overload error	These errors mean the model platform is busy. Your task is not broken. Retry the run. Occasional retries are a normal part of trial work.
Grading fails with an error, or a run comes back without a score	The grader scores each run three times and needs at least two valid samples to produce a score. When it gets fewer than two, grading fails for that run. Rerun the trial. Transient grading errors usually clear on a retry.
Every run scores near the top of the 0.0 to 1.0 scale	High scores across all runs usually mean the failure you intended never fired. This is a property of the task, not a tooling problem. The What Makes a Failure Meaningful section explains how to diagnose it and what to change.
The score is 0.0 on every run	Check your holistic rubric for factual errors. On a task started before the rename, the file is tests/grader-guidance-consolidated.md . The grader may be penalizing correct agent behavior against wrong ground truth. The Writing the Holistic Rubric section covers how to state accurate ground truth.
reward-correctness.txt reads N/A on every run	This is by design and needs no action. Under the Grading Standard, reward-correctness.txt reads N/A on every run by design, because correctness is evaluated inside the criteria rather than as a separate score.
Changes to the workspace do not appear in trials	The trial bakes the workspace into its environment image when the image is built. Regenerate the patch as described in the Workspace & workspace.patch section, then rerun with --force-build so the image is rebuilt with your changes.
The error workspace/: No such file or directory	Run bash scripts/build-workspace.sh <your-task-slug> first. This command builds the task workspace from your pinned commit and patch. The Workspace & workspace.patch section explains how the workspace is built.
Unsure whether editing the holistic rubric requires --force-build	It does not. The grader reads the holistic rubric fresh on every grading pass, so a plain rerun picks up your edits. The --force-build flag rebuilds the environment image and is only needed for environment changes such as the Dockerfile or the workspace. The Reference Runs section explains when a rubric edit calls for regrading existing runs.
submit-task.ts reports placeholder text	One of your files still contains default scaffold text. Search instruction.md and your holistic rubric for the phrase Replace this and replace it with your real content.
The agent runs out of context, or inputs look truncated	The trial agent runs the latest Opus model with a very large context window. A failure that happens because the agent genuinely exhausts its context during a trial is valid signal. Mechanical truncation of your task inputs by the tooling is a tooling issue, not a task defect. If your inputs appear truncated, retry the run, and report the problem in the project Slack channel if it persists.

The agent refuses or gets overly cautious on a security task	Rephrase the prompt so the legitimate engineering intent is explicit, for example by naming the defensive goal of the review. If refusals persist across trials, post the task in the project Slack channel.
<code>stage-atomic-rubric.ts</code> stops because the task carries two atomic rubric files	A task carries one atomic criteria file: <code>tests/atomic-rubric.yaml</code> on new tasks, or <code>tests/rubrics.yaml</code> on tasks created before the current release. When both exist with different content, staging stops on purpose. Keep the file your task was created with, remove the accidental copy, and never rename a committed task file.
Packaging stops because fewer than four copied runs are clean	This stop appears when more than four runs are copied and fewer than four of them are clean. Re-run the failed trials or remove the broken copies from <code>reference-runs/</code> , then package again. A verifier-side timeout counts as clean. See Reference Runs .
Scripts fail with permission denied on WSL	Files sometimes lose their executable bit when the zip is extracted on WSL. Run <code>chmod +x scripts/*</code> from the toolkit root and retry.
A Rails container fails to build with a bootsnap error	Add <code>DISABLE_BOOTSNAP=1</code> to your <code>.env</code> file, then re-create the container.
The container build fails on a heredoc or reports an unknown Dockerfile instruction	Update Docker Desktop. Recent versions install BuildKit, which the build needs for heredoc support.
npm or esbuild reports that it was installed for another platform	The script was run on your host machine instead of inside the Authoring container, so two architectures are being mixed. Run toolkit scripts inside the container. If it happened inside the container, run <code>npm install</code> and retry.
A trial fails with a certificate error such as <code>UNKNOWN_CERTIFICATE_VERIFICATION_ERROR</code>	Check <code>task.toml</code> . This happens when <code>allow_internet</code> was set to <code>false</code> . Set it back to <code>true</code> ; the trial needs the network to reach the model.
The container build stops at <code>could not read Username for 'https://github.com'</code>	One of GitHub's edge servers intermittently returns 401 over HTTP/2. Run <code>git config --global http.version HTTP/1.1</code> and retry the build.
The extracted toolkit is missing <code>explore/repos</code> or has broken links	Re-extract the zip with <code>unzip</code> or 7-Zip rather than the operating system's built-in extractor, which drops symbolic links.