

Raccoon Docs

Raccoon is a Surge AI project where software developers create realistic tasks to help evaluate and improve coding agents.

Learn what the work involves and find the instructions, tools, and reference material you need to author a task.

New to Raccoon?

Required before starting: read the essentials of confidentiality, acceptable AI use, and reporting your time. Then learn what you'll create and how a task comes together.

Before you start →

Recent announcements

September 16, 2026

Six toolkits are deprecated: human-essentials, awbw, stocks-in-the-future, community-foundation, endsideout, and flaredown.

Do not start a new task on one. They stay in the project so tasks already created on them can be finished, revised, and reviewed, so if you have one in progress or returned for...

September 14, 2026

A new toolkit release, 4ead3c97eb, is available, and every task you have not yet submitted moves onto it.

Start every new task on it. Download it from your task page and follow Set up the toolkit. Move every task you are still working on, including one in reviewer feedback, onto this...

September 11, 2026

You can change dependency versions in workspace.patch.

The trial image installs dependencies after applying your patch, so a package you add, upgrade, downgrade, or remove in a manifest is what the trial actually gets. This makes a...

All recent announcements

Working on a task?

Authoring overview

Find the steps from exploring with an agent to preparing your submission.

Meaningful failures

Decide whether an agent's mistake is meaningful enough to build a task around.

Find commands and task files

Look up scripts, required files, and when to use them.

Troubleshoot a problem

Get help with setup, detectors, grading, and packaging.

Open project tools and resources

Find the toolkit repositories and other project links.

Before you start

Required reading for every participant. Read and follow these requirements before starting work or downloading or opening project materials. The linked policies give the full details.

Confidentiality

Keep project materials on your local machine and within the project's approved tools, submission workflow, and communication channels. Do not upload or share them outside the project, including in private repositories or unapproved AI tools. Do not bring AI note-takers or recording tools to Office Hours. Everything you create for the project, or with project tools and tokens, is work made for hire owned by DataAnnotation.tech. Do not use project resources for personal projects.

Read the full confidentiality requirements.

Acceptable AI use

Use AI to support your thinking; you remain responsible for everything you submit. Review everything it drafts, verify its claims, and apply your own engineering judgment. An AI assessment does not replace your human verification of the failure. Short rationales, explanations, and feedback must be in your own words.

Read the acceptable AI-use policy and examples.

Reporting time

Report each day's hours before signing off; if you miss a day, report them at the start of your next session. Report active project work, including onboarding and revisions. Time spent only waiting for runs or reviews is not reportable. You can report time against an open task without submitting it first.

Read what time counts and where to report it.

Continue

[Understand the project](#)



Understand the project

Required before starting: confidentiality, AI use, and time reporting

Read Before you start before beginning Raccoon work or downloading or opening project materials. Every participant must follow these requirements throughout their work on the project.

Raccoon is a Surge AI project where software developers create realistic tasks to help evaluate and improve coding agents. As a task author, you use your engineering judgment to find mistakes that matter, turn the situations that exposed them into repeatable tasks, and explain how an agent’s work should be judged.

What you’re creating

A **task** is an engineering request packaged with everything needed for a coding agent to attempt it and for its response to be evaluated.

You work with a provided **toolkit**: a package containing one or more software repositories and the tools to explore with an agent, author tasks, run and grade trials, and package submissions. Each task targets one repository.

The available toolkits cover different technology stacks and business domains. Unless your task page assigns a repository, choose one whose language and domain are familiar to you. That familiarity helps you identify realistic work and recognize when an agent makes a consequential mistake.

You come up with the request. Put yourself in the shoes of an engineer at the company whose codebase you’re working in: what would they reasonably need to do? Draw on your own experience and what you learn from the repository. You might ask the agent to build a plausible feature, fix an existing bug, or investigate and resolve an operational problem.

A **trial** is one attempt at a task, followed by grading. The toolkit runs multiple trials from the same starting situation so you can inspect the results and capture evidence of the meaningful failure.

Your final submission is one archive with four parts:

Part	What you prepare
Instruction	A prompt that tells the coding agent what engineering work to do.
Starting environment	The repository state and context the agent receives before attempting the request.

Part	What you prepare
Grading guidance	Verified facts and criteria for judging the agent’s work.
Evidence	Saved trials showing how the task behaves, along with current automated detector reports.

The coding agent that attempts the task is called the **trial agent**. A separate LLM, the **grader**, evaluates its work using the project’s Grading Standard and your task-specific guidance. The grader does not attempt the task itself. Your guidance helps it recognize correct behavior, understand the consequences of mistakes, and score stronger work above weaker work.

See Task files for the contents of the archive and How grading works for the evaluation process.

Why meaningful failures matter

A useful task reveals a weakness that would matter on a real engineering team. An agent might leave a security vulnerability, deliver an incomplete feature, make an incorrect diagnosis, or claim to have verified something it never checked.

A **meaningful failure** is a mistake with a real consequence. Finding one gives you a specific behavior to investigate and reproduce. The finished task needs reference-run evidence of that failure. Successful attempts are useful too, and the grading should reflect the quality of each response.

The request must be realistic and fair: the agent needs access to the information and tools required to succeed. A failure caused by an ambiguous request, an artificial trap, or a broken environment does not establish a weakness in the agent’s engineering ability. A response that merely makes you rephrase your request does not meet the project’s failure bar.

Use Meaningful failures to assess a candidate before investing in a full task.

An example: permissions after a role change

Consider a hypothetical task in ZenBill, the payment and invoicing platform available in the Raccoon toolkits. Start a new conversation with the repository at the toolkit’s pinned commit—the specific version of the code supplied as the starting point.

Suppose you ask the agent to add a way for a company administrator to change a teammate’s role. In this proposed feature, demoting an administrator to a read-only role should remove their

ability to initiate payments. The agent implements the role change but fails to revoke that permission. You inspect the agent's changes and verify in the local application that the demoted user can still initiate a payment. That is a consequential mistake in the agent's implementation.

That observation becomes the basis for a task:

1. **Preserve the starting situation.** Keep the original prompt and the toolkit repository's pinned commit. Each trial begins with a new conversation at that same commit, before any of the agent's changes.
2. **Explain how to judge the work.** Record the intended permission behavior, the verified repository facts, and how to check whether a response handles role changes correctly. Put this guidance in the grading material; keep answers and corrective hints out of what the trial agent receives.
3. **Test reproducibility.** Use the toolkit's trial-running script to launch four trials concurrently. Inspect the agent's work and grades for each trial. Look for evidence of the permission mistake in the saved runs. Attempts that correctly remove the permission should receive better grades for that behavior.

The original mistake helps you discover the task. The saved trials provide evidence that the finished task can reproduce the failure, and the grading guidance lets it evaluate future responses too.

From idea to submission

The authoring overview gives the detailed sequence. At a high level, there are four phases:

1. **Explore.** Set up the toolkit and choose a repository. Use the agent to understand the codebase, investigate existing bugs, and identify plausible features or other useful work. Develop a realistic request, ask the agent to attempt it, and investigate any meaningful mistake you observe.
2. **Preserve and author.** Capture the starting situation and prepare the instruction and grading guidance. The starting-point guide explains how to use a snapshot or prepare the task manually.
3. **Test and grade.** Use the trial-running script to launch four concurrent trials. Inspect every result, confirm the saved runs include evidence of the failure, and check that the grades make sense. Save selected trials as **reference runs**. Run the **detectors** at the required checkpoints.
4. **Submit.** Package the verified task, submit it through Surge, and respond to review feedback.

You write the initial grading guidance as a **holistic rubric**: the task context, verified facts, and guidance for evaluating the response across the project's grading dimensions. During testing, you generate an **atomic rubric** of smaller, independently judged criteria and regrade the saved runs. Check that both formats fairly reflect the work before submitting. The holistic rubric and atomic rubric guides explain those steps.

What makes a good task

A good task has all of these qualities:

- **A request grounded in the repository.** The engineering work makes sense for the code and starting state you provide.
- **A consequential mistake.** The failure is verified and has an impact that would matter in real software work.
- **A fair starting point.** The agent can discover what it needs to succeed, without being given the answer or corrective hints.
- **Evidence from repeated trials.** The selected reference runs demonstrate the meaningful failure. An exploratory observation must be supported by evidence from the finished task.
- **Accurate grading.** The guidance is based on verified facts, accepts valid approaches, and makes stronger work score above weaker work in both grading formats.

The failure criteria and reference-run guide explain the requirements in detail.

Where to start

Ready to create your first task? Start with Your toolkit and workspace to see how the tools and files fit together. The authoring overview then takes you through developing a task, grading, and submission. You can return to the linked guides when you reach each stage; you do not need to learn every command or grading rule in advance.

Next step

[Your toolkit and workspace](#)



Meaningful failures

A **meaningful failure** is an agent mistake with a real consequence in realistic engineering work. You should be able to explain what the agent did wrong, verify why it was wrong, and show why it matters.

First assess the mistake itself. Then run trials of the finished task and save reference runs that include evidence of the meaningful failure. An exploratory observation alone does not establish what happened in the finished task.

How serious is the mistake?

The behavior must meet all four criteria:

1. **Broad agreement.** At least 80% of senior software engineers would agree it is a mistake. Judge whether the evidence supports that level of agreement, rather than relying on a personal preference for a particular approach.
2. **Feedback worth giving.** You would give a teammate corrective feedback for the same decision.
3. **Serious enough to block.** You would block a pull request over it. For work that produces an analysis or recommendation rather than a code change, apply the same standard: would you stop that work from being used until the mistake was addressed?
4. **A real consequence.** Explain the impact, such as corrupted data, an incomplete feature users rely on, misdirected money, or unauthorized access.

An answer that merely makes the requester rephrase and try again does not meet this bar.

A failure can happen before any code is written. Fabricating a test result, giving a consequentially wrong diagnosis, or concealing incomplete work can matter as much as a code defect. The Grading Standard dimensions describe the broader range of engineering behavior we evaluate.

Examples of meaningful failures

These examples illustrate the behavior and its consequence; verify both in the repository you are working with.

Retained permissions

The agent implements role changes but leaves an administrative permission active after a user is demoted. The demoted user can still initiate a payment that their new role should prohibit. The agent's change creates an authorization vulnerability.

Incomplete rollout

The request asks the agent to show an invoice's payment due date in the dashboard and reminder emails. The agent updates the dashboard, omits the emails, and reports the feature as complete. Customers relying on those reminders still receive no due date and may miss the payment deadline.

Rebuilding instead of diagnosing

Asked why an endpoint returns `null`, the agent fails to find the existing endpoint and creates another implementation. The application still calls the original endpoint, so the reported problem remains unresolved. The duplicate also introduces competing implementations for future maintainers to reconcile.

Incorrect result

The agent produces a polished report of outstanding invoice balances but counts already-paid invoices as unpaid. The resulting totals are wrong and would lead the team to pursue payments customers have already made. A professional-looking response does not compensate for an incorrect result with a real consequence.

What does not count

A reasonable interpretation of an ambiguous request

The rubric expects a field rename to affect only migration files, but the request could reasonably be understood to include corresponding application-code changes. An unstated preference in the rubric does not make the agent's interpretation a meaningful failure.

A necessary clarifying question

Before changing payment behavior, the agent asks which users should be allowed to initiate a transfer because the request leaves that decision open. Asking for information needed to make a safe, correct change is sound engineering judgment.

A problem caused by the evaluation setup

A run stops because of an imposed tool-call time cap, or the agent cannot use a command available only in the Explore container. Those limitations do not establish a weakness in the agent's engineering behavior.

Judge the cause, not just the symptom. A port mismatch caused by the evaluation setup is different from an agent misconfiguring the application despite having the necessary information. Broken builds, command errors, and failures discovered during testing can be meaningful when they reflect the agent's decisions and meet the seriousness criteria above.

Verify the mistake

Build a clear chain of evidence:

1. **What was requested?** Check that the request makes sense for the supplied repository and that the agent could discover what it needed to succeed.
2. **What did the agent do?** Inspect the actual response, code changes, and relevant actions. Suspicious code or the agent's description alone is not proof.
3. **Why is it wrong?** Verify the expected behavior against the code and relevant project context. Behavior that is intentional is not a bug simply because it looks unfamiliar.
4. **What is the consequence?** If you claim the application behaves incorrectly, run it and check that behavior yourself. Reading the code or relying on the agent's description is not enough. For analysis or reports, verify the claims against the underlying evidence.

Be specific about what you verified and any limits on verification. For a security claim, establish who can perform the action, under what conditions, and what access or impact results. See Security tasks for the guidance for security contractors.

Show that it is reproducible

At least a quarter (25%) of the saved reference runs must demonstrate the meaningful failure. Run the task from the same starting situation and inspect the results; the failure does not need to occur in every run. See the reference-run requirements for the submission details.

A low score alone does not demonstrate the failure. Inspect what the agent actually did in each run and identify the behavior that meets the definition above. Successful runs can be included, and the grading should reflect the quality of each response. No particular score distribution is required.

The reference-run guide explains how to launch trials and save the evidence. If none of the saved runs demonstrate the failure, use the trial troubleshooting guidance to investigate before submitting. Do not add unsupported penalties to manufacture low scores.

Confidentiality

All materials provided through Raccoon are confidential and may be used only for this project. These requirements also apply to materials you create from project information, including notes, screenshots, and task submissions.

Use and disclosure

- **External sharing.** Project materials must not be shared with anyone outside the project or uploaded, mirrored, forked, or posted to outside services. This restriction applies to private repositories and accounts as well as public ones.
- **AI tools.** Project code and documents may be used only with the AI tools the project explicitly directs you to use. They must not be sent to other AI services. See [Acceptable AI use](#) for permitted assistance and human verification requirements.
- **Toolkits and credentials.** Your toolkit and API key must not be shared.
- **Model access.** Use the model access supplied through the project proxy only for work that supports a Raccoon task.
- **Reviewer assessments.** The assessment, including its content and answers, must not be discussed anywhere, including in project Slack.

Outside services include code-hosting platforms, forums, social media, chat services, paste sites, personal websites, video platforms, and public documents. Restricting access to a private account does not authorize sharing project materials there.

Storage and communication

Keep task materials on your local machine. Submit task archives through the designated Surge submission workflow. Project discussion belongs in the approved Slack workspace.

The signup link in the logbook is confidential. For referrals involving people already on the platform, contact an admin by direct message.

Confidential materials

These requirements cover materials you receive through the project, work you produce, and information derived from either. Examples include:

Category	Examples
Code and environments	Repositories, toolkits, devcontainers, snapshots, branches, diffs, and workspace patches.
Tasks and evaluations	Prompts, holistic and atomic rubrics, detector reports, reference runs, reviews, reviewer assessments, and example submissions.
Project resources	The Task Catalog, Grading Standard, Workflows, FAQ, linked documents, and office-hours materials, recordings, notes, and transcripts.
Materials you create	Screenshots, screen recordings, walkthroughs, notes, and write-ups about project repositories or toolkits.

Meetings and recordings

The team records office hours and makes the recordings available through project links. Do not bring AI note-takers or recording tools, such as Granola, Fireflies, or Otter, to office hours.

Model proxy use

Traffic through the project proxy is logged. Misuse of project model access may result in penalties or removal from the platform.

Work made for hire

As a reminder, you signed an agreement when you joined us. Anything you create for us or using our tools is a "work made for hire" that is owned by DataAnnotation.tech. That includes using our tokens (or any other technology to which we give you access) for a personal project. Stop and think whether the risks are worth it before you use our resources to build that app or video game project.

Questions about sharing

If you are unsure whether material may be shared or a destination is approved, ask in [#ext-surge-raccoon-questions](#) before moving or transmitting it. Until you have confirmation, treat the material as confidential and do not share it.

Acceptable AI use

Use AI to augment your own thinking, never to replace it. Raccoon exists to advance what AI can do. Your engineering judgment is what makes the work valuable, and you are responsible for everything you submit, regardless of which tools helped produce it.

Where AI can help

AI is allowed throughout much of the work:

- **Repository exploration.** Use Claude or Codex to explore and understand the repositories, and use the toolkit's skills.
 - **First drafts.** Have AI put your thinking into words. The ideas and judgment must be yours.
 - **Task artifacts.** AI can create artifacts such as a seeded design specification or an agent-written diff when you deliberately design a realistic scenario grounded in your own expertise.
 - **Copy editing.** Using a model to catch typos and grammar is allowed.
 - **General questions.** Outside assistants such as ChatGPT or Gemini can help with general questions, provided you double-check their answers. Never paste project code or documents into tools the project has not directed you to use. Follow the confidentiality requirements.
-

Review everything AI drafts

Do a human pass on every AI draft. Verify each claim, remove repetition and unclear prose, and retain useful precision such as verified file and line references.

Unchecked AI output does not meet the quality bar. Examples include rubrics containing factual or judgment errors and prompts that no real engineer would send given the repository. People have been removed from the project for this kind of AI misuse.

Write rationales and feedback yourself

This project, side projects you might be asked to work on, and questions in Slack sometimes ask for a few sentences of reasoning, such as the rationale for a rubric change. Those sentences are how the team sees your reasoning, so write them yourself. Asking AI to correct your language or to translate a draft you wrote is fine; having it write the rationale is not. If you used AI to correct or translate, include your original text as well.

Verify the failure yourself

Do not use Claude to verify Claude. Your human verification of the failure is the point of the task. Another AI assessment does not establish that the failure is real; inspect the work and verify it against the repository or observed application behavior.

See Meaningful failures for how to assess the evidence.

When you are unsure

If you cannot tell whether a use supports or replaces your thinking, do not use AI for it. Ask in [#ext-surge-raccoon-questions](#) first. The team would rather clarify the boundary than remove someone: borderline cases receive feedback, while egregious cases can result in removal without warning.

This policy applies to Raccoon. If another platform project prohibits AI, that prohibition includes using it to augment your work.

Your toolkit and workspace

You'll work with a coding agent in a terminal, using a toolkit provided through Surge. This page introduces the tools and files you'll encounter before you start the setup instructions.

Read [Before you start](#) for the required confidentiality, AI-use, and time-reporting guidance before downloading or opening project materials.

Choose familiar work

Usually, you choose from the toolkits and repositories available in your task page. Look for a language, business domain, and area of engineering you know well—whether that is frontend, backend, or another specialty. If your task page assigns a repository, use that one.

Each **toolkit** packages one or more software repositories with the tools for authoring and evaluating tasks. Each task you create targets one repository. Your familiarity with that codebase's stack and domain will help you identify realistic requests and recognize consequential mistakes.

The repository catalog describes the available codebases. The repository selection guide will help you make your choice.

Surge and your local workspace

Your task page in **Surge** gives you access to toolkit downloads and the project's model connection details. You return to Surge to submit the completed task and respond to review feedback.

After downloading and extracting your chosen toolkit, you work locally. You can browse and edit files in VS Code or another editor, or work entirely through a terminal and coding agent. The common starting point is a terminal session running **Codex** or **Claude Code**, where you explore the repository and give the agent engineering requests.

As you work, keep three things distinct:

- **The application repository:** the company's codebase that you and the agent explore. The agent's implementation changes happen here.
- **Your task folder:** the prompt, starting-state information, and grading guidance that define the task another agent will attempt.
- **Trial results:** records of what an agent did when it attempted that task, together with its grades and other evidence.

The toolkit provides scripts to turn your work into a task folder, run trials, and package a submission.

Two environments for different stages

The toolkit uses two development containers. A **container** provides the software and dependencies needed for that stage of work.

Explore is where you get to know the repository, run the application when available, and interact with the coding agent. You use it to develop a request and investigate the agent's response.

Authoring is where you prepare the reusable task, run trials, check the results, and package the submission. Task-building and trial commands run from the toolkit root inside this container.

You begin in Explore and switch to Authoring when you are ready to build the task. The instructions for each stage tell you which environment and directory to use. You can use your preferred editor for text changes; trials and packaging must run inside Authoring.

Explore provides the coding agent and snapshot tools. Authoring provides the tools for building tasks, running trials, grading, and running detectors. Switch to Authoring when you are ready to turn a captured failure into a task.

The files you'll work with

You don't need to learn the complete directory layout before starting. These are the main parts of a task folder:

		text	Copy
harbor-tasks/<your-task>/			
├ instruction.md	The prompt you prepare for the agent		
├ task.toml	The task's configuration		
├ environment/	Information for recreating the starting state		
├ tests/			
│ └ holistic-rubric.md	The grading guidance you write and verify		
├ reference-runs/	Selected trial results copied by a script		
├ rubric-regrades/	The atomic grade of each reference run		
└ detectors/	Reports produced by detectors		

This is a simplified view. The task-building tools create the task structure; you inspect the files and prepare the task-specific content. `instruction.md` and `tests/holistic-rubric.md` are the two main documents you author. Other files configure the task or preserve the starting situation.

The tooling also generates grading files, trial results, and **detector reports**—automated checks for task-authoring problems. Some supplied files are shared infrastructure and must not be edited. The stage

guides explain what to change, and Task files provides the complete reference.

What Harbor does

Harbor runs and evaluates the task through the toolkit's supplied scripts. A trial starts from the task's saved environment, gives the instruction to the coding agent, and produces the agent's response and grading results.

You interact with Harbor through the commands in these docs, such as [scripts/harbor-run](#) . You do not need to learn its internals. Your work is to launch the documented command, inspect what the agent did, and check that the grade reflects the response.

Trial output is written under [harbor-jobs/](#) . When you select trials as evidence, another script copies them into the task's [reference-runs/](#) folder. Grading those runs under the atomic rubric stores each grade in [rubric-regrades/](#) for you. The packaging script then creates the archive you submit through Surge.

Next step

[Choose a repository](#)



Choose a repository

Usually, you choose your own repository. If your task page assigns one, use that repository and continue to Set up the toolkit.

Otherwise, choose from the repositories available in your task page. Look for a familiar language, business domain, and area of engineering, such as frontend or backend work.

A few toolkits are deprecated and kept only for tasks already created on them. Do not start a new task on one; see Toolkits available only for existing tasks.

1. Review the repository catalog for a toolkit-level comparison, and the full repository appendix for every repository's language and purpose in one table.
2. Prefer a codebase where you can run or inspect the relevant behavior yourself.
3. If a toolkit contains multiple repositories, choose one for your task. The task-running system supports only one repository per task, so the work must be contained within that repository.

The Task Catalog can help you see which areas already have coverage. Checking it is optional; its task counts do not tell you whether a specific idea has already been used.

Repository tours are available for ZenBill and Palolo on Links and resources.

The repository catalog also describes toolkits containing multiple repositories and the Zeta reference corpus.

Next step

[Set up the toolkit](#)



Set up the toolkit

These steps configure the toolkit and check that the repository and coding agent run in the Explore container.

Read [Before you start](#) for the required confidentiality, AI-use, and time-reporting guidance before downloading or opening project materials.

System requirements

- **Memory:** at least 16 GB of RAM.
- **Docker:** install and start Docker Desktop with at least 4 GB allocated to Docker.
- **Windows:** use WSL 2 and keep the extracted toolkit in the native WSL filesystem, not under `/mnt/c`.
- **Storage:** allow space for containers, repositories, and trial results.

macOS and Linux provide the smoothest setup. Intel Macs have known container limitations; ask in the project questions channel if normal Docker troubleshooting does not resolve them.

Download the toolkit

Download the toolkit for your chosen or assigned repository through your task page. Use the download and model connection details provided there.

Use the latest announced toolkit version for each new task. If your copy is already current, you do not need to download it again. Keep an in-progress task on its existing version unless the migration instructions direct otherwise.

Configure the toolkit

Read the toolkit [README.md](#) and the latest entry in [CHANGELOG.md](#).

Create `.env` at the toolkit root using the values shown on your task page. It must contain:

dotenv

Copy

```
ANTHROPIC_API_KEY=...  
ANTHROPIC_BASE_URL=...  
USER_ID=...
```

Despite the `ANTHROPIC_` prefix, these variables configure access to the project's model proxy. The toolkit uses the same key and base URL for Codex and other supported models, including those from other providers. Keep these variable names when using Codex.

Your API key is personal. Do not share it or include it in task files, detector reports, or Slack messages.

Start the Explore container

From the `explore/` directory:

bash

Copy

```
npx @devcontainers/cli up  
npx @devcontainers/cli exec bash
```

The first build takes several minutes. To replace a broken container with a clean build:

bash

Copy

```
npx @devcontainers/cli up --remove-existing-container --build-no-cache
```

VS Code users can instead open `explore/` with the Dev Containers extension and reopen the folder in the container.

Repositories in a polyglot toolkit

For a toolkit with repositories under `/workspace/repos/`, initialize your chosen repository inside Explore:

bash

Copy

```
run-app <repo>  
cd /workspace/repos/<repo>
```

Replace `<repo>` with the repository's name. `run-app` checks out the code and prepares its dependencies. Run it even for a repository without a web application.

One application runs at a time. To switch repositories:

bash

Copy

```
run-app --stop
run-app <other-repo>
cd /workspace/repos/<other-repo>
```

Local sign-in

Follow the toolkit README for repository-specific setup and prepared account details. These repositories use the following local sign-in routes:

Repository	Local sign-in route
Breezy	<code>/pro_signin</code>
Potion's <code>potion-app</code>	<code>/auth/login</code>

Check the README for which integrations are available locally and how missing provider credentials are handled. Being able to start the app does not establish that an external integration works.

Verify the setup

Inside the Explore container, confirm that:

- the target repository is present;
- its application or test suite works where supported (use `run-app` when applicable); and
- `codex` starts successfully.

Codex is the default trial agent for new tasks. See Trial agents for Claude Code, browser access, and authentication help.

Next step

Authoring overview

→

Authoring overview

Required before starting: confidentiality, AI use, and time reporting

Read Before you start before beginning Raccoon work or downloading or opening project materials. Every participant must follow these requirements throughout their work on the project.

Before starting, choose a repository and set up the toolkit.

Explore and find a task

Use the agent to understand the codebase, investigate bugs, and identify useful work. Develop a request that an engineer at the company could reasonably make, then inspect the agent's attempt.

When you observe a mistake, verify it and its consequences using evidence from the repository and the agent's work.

Develop a task →

Reference: Meaningful failures

Build the task

Preserve the request, repository state, and any conversation context from before the agent's attempt. Switch to Authoring, create the task folder, and verify that its environment recreates that starting point.

Review the task prompt and write the holistic rubric: the verified facts and guidance the grader needs to judge the response. Keep answers and corrective hints out of what the trial agent receives.

Capture the starting point →

Also in this stage: Prepare the task environment · Review the task prompt · Write the holistic rubric

Run trials and review holistic grades

Run the detectors, then use the toolkit script to launch trials. Each trial runs the agent and grades its response using your holistic rubric. Inspect the agent's work and check that the grade is justified.

Save clean trials as reference runs, including evidence of the meaningful failure. If the task or grading needs changes, update it and refresh the affected results before continuing.

Run and review trials →

Reference: Pre-trial detectors

Reference: What to rerun after edits

Generate the atomic rubric and its grades

Generate an atomic rubric from the holistic rubric: smaller criteria that the grader judges individually. Grade the saved reference runs under it, check that the two grading methods reach consistent conclusions about the agent's work, and store the atomic grades in your task.

Generate the atomic rubric and its grades →

Package and submit

Complete the final checks and use the packaging script to create the task archive. Submit it through Surge with enough context for the reviewer to understand the failure and your evidence.

Respond to feedback and keep revisions attached to the same task. The task is finished when a reviewer accepts it.

Submit for review →

Next step

[Develop a task](#)



Develop a task

Explore the repository with the coding agent to understand how the application works and identify useful engineering work. Use what you learn to develop a realistic request, then inspect the agent's attempt.

Choose a trial agent before you begin. The agent you use during exploration will also run the task's reference trials.

Get familiar with the codebase

Work in the Explore container. Use the agent to trace how a feature works, investigate unexpected behavior, or understand an unfamiliar part of the code. You can also browse the repository in your editor.

Use `run-app` when the repository has a runnable application. Confirm the starting behavior yourself rather than relying on the agent's description.

Choose an engineering request

Put yourself in the shoes of an engineer working on this codebase. Choose work that fits the application and its business domain, such as implementing a feature, fixing an existing bug, or resolving an operational problem. Tasks can also ask the agent to review a code change or investigate a technical question, where its conclusions can be checked against the repository.

If your assignment is to contribute to an existing project, work on its normal engineering tasks with the agent. You are not expected to spend all your time looking for failures. When you encounter a meaningful failure, use the steps below to turn it into a Raccoon task.

For security contractors, prioritize security and abuse-resistance work. See Security tasks for focus areas and approaches.

You can introduce a bug or prepare a scenario, provided the resulting problem is something an engineer could realistically encounter.

You can use the Task Catalog to see coverage by repository and subsystem while choosing an idea. It does not show whether your specific task has already been done.

Write the request as a prompt and give it to the agent.

Review the agent's work

Review the agent's investigation, code changes, verification, and final response. Look for consequential mistakes in both what it does and what it says—for example, code that breaks existing behavior, an incorrect explanation, or a confident claim that its work is complete when it hasn't verified the result.

A meaningful failure is a mistake that would matter to an engineer relying on the agent's work. See Meaningful failures for the criteria and examples.

When you find a meaningful failure

Check the agent's work against the repository. If you claim the application behaves incorrectly, run it and confirm the problem yourself. Establish what went wrong, what should have happened, and why the difference matters.

To turn that attempt into a task, preserve the prompt and repository state from before the agent's response. If you're using a snapshot, run your agent's snapshot command while that response is still the latest turn. Capture the starting point explains the process and how to build a task manually when no conversation history is needed.

A snapshot alone is not a submission. Continue through the remaining authoring steps: build the task, write both rubrics, run trials and the required detectors, and submit the completed task for review.

You may try several requests before finding a meaningful failure. That's a normal part of developing a task.

Next step

[Capture the starting point](#)



Capture the starting point

Each trial needs to begin where the agent's failed attempt began: with the same prompt, relevant conversation history, and repository state. The failed response itself is excluded so the trial agent can make its own attempt.

Use a snapshot when earlier conversation is needed to reproduce the behavior. When one instruction and a prepared repository state are enough, you can create the task manually. Both approaches produce the same kind of task archive.

Capture a snapshot

Use English throughout the conversation you plan to capture; the recorded session ships with your task.

Keep your task setup changes uncommitted. The task's pinned commit must already exist in the supplied repository; your setup changes are carried in the workspace patch.

In your Claude Code or Codex session inside Explore, run the appropriate snapshot command while the failing exchange is still the latest turn:

text Copy

```
# Claude Code
/create-snapshot:snapshot

# Codex
$snapshot
```

The snapshot flow asks what went wrong, what the correct answer is, and why it matters. Provide that explanation through the snapshot flow; it supplies context for authoring the task.

Keep answers out of the recorded conversation

Future trial agents receive the recorded conversation. Keep the failure explanation in the snapshot annotations, separate from the conversation supplied to those agents. That conversation must not contain the answer or corrective hints.

If you sent more messages after the failure, follow If you continued past the failure before capturing it.

If you continue working in the same Claude conversation after snapshotting, rewind immediately so a later snapshot does not record the earlier snapshot command. A session containing that command is rejected during packaging.

Start the Authoring container

Switch to Authoring to convert your snapshot or create a task manually. Exit the Explore container's shell. In a terminal on your host machine, go to the toolkit root and run:

```
npx @devcontainers/cli up  
npx @devcontainers/cli exec bash
```

bash

Copy

You can edit files in any editor. Run task-building commands, trials, and packaging from the toolkit root inside Authoring.

Convert the snapshot

In the terminal inside Authoring, run this from the toolkit root, replacing `<snapshot>` with your snapshot directory's name:

```
npx tsx scripts/snapshot-to-task.ts --snapshot explore/snapshots/<snapshot>
```

bash

Copy

Conversion creates the task folder. Earlier conversation turns become `environment/session.jsonl`, the prompt that produced the failure becomes `instruction.md`, and the repository changes needed for the starting state become `environment/workspace.patch`.

Inspect the generated task

Open the generated files and verify that:

- `instruction.md` contains the intended prompt.
- `environment/session.jsonl` stops before the failing reply and contains no answer or strong hint.
- The session and instruction describe the same situation.
- `environment/workspace.patch` recreates the repository state from before the agent's attempt, without including changes the agent made during that attempt.
- The recorded session does not contain a snapshot command.

If the only unwanted turn is the snapshot command itself, remove that command from `environment/session.jsonl` and inspect the session again. If the conversation contains the answer, corrections, or strong hints, return to Explore and create a clean snapshot rather than deleting selected lines.

Next, prepare and verify the task environment.

Create a task manually

Start the Authoring container, then copy the scaffold from a terminal at the toolkit root. Replace `my-task-slug` with a unique name for your task:

```
cp -r harbor-tasks/_task-scaffold harbor-tasks/my-task-slug
```

bash

Copy

For a multi-repository toolkit, select the supplied Dockerfile for the repository your task targets. List the available files, then copy the matching one, replacing `<repository>` with its filename suffix:

```
ls task-shared/Dockerfile.*
cp task-shared/Dockerfile.<repository> harbor-tasks/my-task-slug/environment/Dockerfile
```

bash

Copy

Single-repository toolkits already include the correct Dockerfile. Use the supplied file without editing its contents.

In `task.toml`, check that the repository and pinned commit match your intended starting point, and set the trial-agent harness. Write the prompt in `instruction.md`. A manual task does not need `environment/session.jsonl`.

Continue to prepare the task environment. You'll write the holistic rubric as part of the following authoring steps.

If you continued past the failure

For Claude Code, use `/rewind` until the prompt that produced the failure is your most recent user message, then snapshot. Do not resend the prompt. If Bash changes from later turns remain in the working tree, revert them manually first.

For Codex, start a fresh session, reproduce the behavior without hints, and snapshot as soon as it occurs. Codex has no project-supported rewind.

A three-turn example

Suppose:

1. You ask the agent to explain a subsystem.
2. You ask it to fix a defect, and its change breaks the application.
3. You ask why the application broke.

If you snapshot at turn 3, turns 1 and 2 become history and turn 3 becomes the task prompt. That captures a request to diagnose the breakage, rather than the defect-fixing request you intended to test.

The snapshot needs to capture the exchange at turn 2. After conversion, turn 1 becomes history and the prompt from turn 2 becomes `instruction.md`. The reconstructed trial workspace should match the repository state when that prompt was sent. Use the agent-specific recovery instructions above to capture that starting point.

Next step

[Prepare the task environment](#)



Prepare the task environment

Your task needs to recreate the repository state the agent should start from. If you captured a snapshot, verify the environment it produced. If you created the task manually, prepare that environment now.

Start with the task folder created in Capture the starting point. Run the commands below from the toolkit root inside Authoring, replacing `<slug>` with your task folder's name.

The starting state has two inputs: the repository commit pinned in `task.toml` and `environment/workspace.patch`, which contains the changes applied to that commit. Changes left only in your local workspace will not reach future trial agents.

Select the target repository

For a toolkit containing multiple repositories, confirm that the task targets the repository where you developed the request:

1. Set `[metadata].repo` in `task.toml` to that repository's name.
2. Use its supplied `task-shared/Dockerfile.<repo>` as the task's `environment/Dockerfile`. Check the file produced by snapshot conversion, or copy the supplied file when preparing the task manually.
3. Run `bash scripts/build-workspace.sh <slug>` to build the workspace and stage that repository's correctness commands.

The workspace build is required for the grader to run the repository's correctness checks. Include any information the task needs from other repositories in the prompt or starting files; the trial agent sees only the selected repository.

If you captured a snapshot

Snapshot conversion already builds the workspace and copies `snapshot.patch` to `environment/workspace.patch`. Confirm that the patch recreates the repository state from before the agent's attempt, without including changes the agent made during that attempt.

If the starting state is correct, continue to Verify the starting environment. If it needs changes, use the steps below to update the patch.

If you need to prepare or change the workspace

First, build the workspace from the task's pinned commit and patch:

Authoring · toolkit root

bash

Copy

```
bash scripts/build-workspace.sh <slug>
```

Make the repository changes needed for the task inside `harbor-tasks/<slug>/environment/workspace/`. Then save the complete difference in the patch:

bash

Copy

```
bash scripts/check-workspace-sync.sh --update-patch harbor-tasks/<slug>
```

For example, you might add a fixture, introduce a bug, or include a diff for a code-review task. Required files and context must be available inside the trial environment. See Trial environment for repository boundaries, excluded files, and how to supply history.

Generate SQL dumps, seed files, and prepared data states from the running application. Writing them by hand can give away the solution. Remove AI commentary from these artifacts; if they do not make sense without it, regenerate them.

Verify the starting environment

Read the patch and remove unrelated lockfile changes, logs, editor files, accidental permission changes, credentials, and unintended binary files. Check supporting files for comments or filenames that reveal the solution.

Rebuild from the saved commit and patch:

bash

Copy

```
bash scripts/build-workspace.sh <slug>
```

Check that the task's premise still holds in this reconstructed workspace and, where applicable, the running application. Confirm that the agent will have the files, dependencies, and context needed to attempt the request, and that its result can be checked locally.

If you change the workspace later, update the patch again. If trials already exist, follow [What to rerun after edits](#) before using those results.

Next step

[Review the task prompt](#)



Review the task prompt

With the starting environment prepared, check [instruction.md](#), the prompt given to the trial agent. This step makes sure the request you developed during exploration is correctly represented in the task.

If you captured a snapshot

Open [instruction.md](#) and confirm it contains the request that produced the failure. Check that it agrees with the recorded conversation and prepared environment. You don't need to rewrite a prompt that already does its job.

If you created the task manually

Put the request you developed during exploration into [instruction.md](#), including any context the agent needs that isn't available in the repository. If you already added it when creating the task folder, review what's there.

Make sure the request is clear

The prompt should read like a realistic engineering request and be written in English. Depending on the task, useful information might include the desired outcome, business context, constraints, or examples of expected behavior.

There is no required structure or length. Use the information the agent needs to understand the job; the examples below are illustrations, not a template to follow.

For security contractors, check the security-task requirements when framing the request.

Give the context, without giving away the solution

Give the agent enough information to understand the job. Requirements and useful context belong in the prompt; the investigation or solution you want the agent to work out should remain part of its task.

For example, a request about permissions could say:

When an administrator is changed to a read-only user, they should immediately lose the ability to issue refunds. Update the role-change behavior and verify that this works for an existing signed-in session.

That states the expected behavior. If the task is meant to test whether the agent discovers why permissions remain active, adding “The authorization cache isn’t invalidated; update the cache invalidation in the role-change handler” would give away that investigation.

Detailed instructions can be appropriate when they are part of a realistic request. The distinction is whether they explain the job or supply the answer to the problem the agent is supposed to solve. Put your verified explanation of the failure and guidance for evaluating the response in the holistic rubric.

Check the prompt against the task environment

Describe the repository state the trial agent will actually receive. If you prepared a bug, fixture, or other change in the workspace, the prompt should match that starting state.

Include any context the agent needs in the prompt or the supplied repository. The task must be possible to complete and evaluate within that environment. For example, a code-review request needs the diff being reviewed, and a request to match expected behavior needs a description or fixture that establishes that behavior. Neither should depend on private context the agent was never given.

Check that supporting files and any recorded conversation do not supply the solution through comments, filenames, or earlier answers. See Trial environment for what the environment includes and how to supply repository history, and Capture the starting point if the recorded conversation needs to be corrected.

Next step

[Write the holistic rubric](#)



Write the holistic rubric

Once the task environment and prompt are ready, write the task-specific guidance for the grading LLM in `tests/holistic-rubric.md`.

The rubric explains what correct behavior looks like, what went wrong in the failure you observed, why it matters, and how to distinguish responses of different quality. The Grading Standard supplies the shared grading rules and dimension definitions; your rubric supplies the facts and guidance specific to this task.

Draft with the toolkit skill

You can write the rubric yourself or use the toolkit skill to help draft it. To use the skill, run this from a Claude Code or Codex session at the toolkit root inside Authoring:

text Copy

```
# Claude Code
/write-holistic-rubric

# Codex
$write-holistic-rubric
```

Provide the task folder, an explanation of the failure you observed, and the evidence you verified. Whether you write the rubric yourself or use the skill, verify its factual claims and review whether its grading guidance is fair.

Read `task-shared/grading-standard.md` and the task's `tests/grader-system-prompt-consolidated.md` to understand the instructions the grader already receives. Keep the rubric in English and refer to “the agent” rather than a particular model.

Older tasks may use an earlier rubric filename. Keep the existing name; see Toolkit updates.

What goes in each section

Include the task-specific context the grader needs directly in the rubric. Don't rely on another document to explain it.

Section	What belongs there
Task context	Briefly summarize the request, relevant subsystem, and necessary background. Aim for two to four sentences.
Business context	Domain concepts needed to judge the response, such as a settlement window or compliance rule. Omit this section when it adds nothing.
Ground truth	Verified repository facts, expected behavior, and evidence the grader can rely on. Include relevant checks, misleading signals, and limits of local verification.
Grading dimensions	Concrete strong and weak behavior under each of the eight shared dimensions. Use the grading dimension definitions to decide where each behavior belongs and explain how it applies to this task. If a dimension has no task-specific guidance, say so.
Heavy penalties	Serious mistakes that warrant a substantial reduction in the grade. Omit this section when it is not needed.

Put repository facts in Ground truth and explain how to judge behavior under the relevant dimension. Avoid repeating the same facts throughout the file.

Verify the repository facts

Check factual claims against the repository and, where relevant, the running application. Cite decisive evidence as `path:line`, and quote short pieces of code when they help explain the behavior.

The grader relies on Ground truth. An incorrect claim there can lead it to misgrade every run, so verify the skill's draft rather than accepting its explanation at face value.

Be specific about which tests and other checks work in the task environment. Before saying a build, test, or check cannot run, confirm that limitation inside a trial. Describe real limitations without excusing verification the agent could have performed. Requirements must be supported by information available to the agent in the prompt, recorded conversation, or supplied environment.

Describe how to judge the response

Explain what makes a response correct, incomplete, or wrong, and why the differences matter. Describe observable behavior and consequences rather than relying on labels such as "good" or "thorough."

Under Narrow Correctness and Broader Correctness, state the deliverable type and the concrete conditions a working result must satisfy. Identify anything a passing test suite would not prove, such as a required behavior the tests do not cover. If a dimension has no task-specific guidance, say so rather than inventing requirements.

Allow valid alternative solutions. An existing implementation or commit may demonstrate one correct approach without being the only acceptable answer.

Give credit for useful progress. An attempt should be able to score higher than a response that only asks clarifying questions, even if the attempt is incomplete or has mistakes.

The failure you observed helps explain the task, but the rubric must apply to future attempts too. Reference runs show how agents behave; they do not define correct behavior. Do not invent requirements to fit a particular run or cite your reference runs in the rubric—the grader does not see them.

Keep meaningful failure cases in the rubric even if none of your reference runs show them. Only the failure your task targets needs to appear in the reference runs.

Keep the distinction between requirements and answers clear. The prompt can state what the agent needs to accomplish. Your verified explanation of the failure, the solution it needs to discover, and the grading rationale belong in the rubric.

Heavy penalties

Use a heavy penalty when a specific serious mistake warrants a substantial reduction in the grade. Explain both the behavior that triggers it and the nearest acceptable behavior that does not.

Example

If the agent claims the tests pass without running them, apply a heavy penalty to Verification & Thoroughness. An explicit statement that the tests were not run does not trigger the penalty.

Describe severity in words rather than specifying a numeric deduction. The heavy-penalty rules explain how to target dimensions or the overall grade without changing the shared scoring rules.

Add checks if the grader needs more evidence

Existing repository tests may not cover the behavior your task is testing. You can add checks that the grader runs against the agent's completed work to provide additional correctness evidence. Additional

checks are optional when existing verification is sufficient.

Keep these grader-only checks outside the trial workspace. Your task's copy of [tests/test-commands.sh](#) is yours to edit for this purpose. See Task-specific test commands for where to put the specs, how to register them, what you can change in that file, and which shared files must not be edited.

Next step

[Run and review trials](#)



Run and review trials

Once the environment, prompt, and holistic rubric are ready, run a batch of trials to see how the task behaves. Review each attempt and its grade, then save reference runs as evidence.

A trial is one attempt by the agent followed by grading. A reference run is a reviewed trial copied into the task's [reference-runs/](#) folder.

Run every command on this page from the toolkit root inside Authoring. Replace `<slug>` with your task folder's name.

Run the detectors

Run the pre-trial detectors, review their reports, and address valid findings before launching trials.

Launch the trials

Launch the first batch of four parallel trials:

bash

Copy

```
scripts/harbor-run harbor-tasks/<slug> --force-build -k 4 --verifier-env GRADER_SAMPLES=1
```

`--force-build` rebuilds the environment image; `-k 4` requests four trials. Use `--force-build` for the first batch and after changing the workspace.

The script runs each attempt and grades it with the holistic rubric. `--verifier-env GRADER_SAMPLES=1` requests one grading sample per attempt. Use `GRADER_SAMPLES=3` instead if you prefer a steadier reward; either count is acceptable for a submission. Results are written under [harbor-jobs/](#), with one directory per trial inside the job directory.

Review each attempt and its grade

Read the agent's investigation, code changes, and final response to understand what it actually did. Check any apparent meaningful failure against the repository or observed application behavior. Then read the grader's explanation and assess whether its conclusions are supported by the work.

For each trial, review [grade.md](#) alongside the agent’s work. It contains the grader’s explanation; [reward.txt](#) contains the overall score from that assessment. See Grading samples for details.

Inside each trial directory, open these files in your editor:

Path	What to review
agent/trajectory.json	The recorded sequence of messages and actions, including the agent’s investigation and response.
verifier/agent-output/	Files the agent created or changed.
verifier/grade.md	The grader’s reasoning and dimension scores.
verifier/reward.txt	The overall 0.0–1.0 score.

A low score alone is not evidence of a meaningful failure, and the runs do not need to fit a particular score distribution.

If something is wrong, follow the guidance for the problem you observed:

- The intended failure did not appear.
- An execution or environment problem disrupted the trial.
- The grade does not match the agent’s work.

Save reference runs

Save at least four reviewed attempts that were not disrupted by environment or execution problems. At least a quarter (25%) of the saved runs must demonstrate the meaningful failure. You may include more than four, and successful attempts are welcome.

See Reference-run validation for how to handle interrupted runs and the exception for a timeout during grading.

Copy the reviewed trials

To save an individual trial:

bash

Copy

```
npx tsx scripts/copy-reference-run.ts harbor-jobs/<job>/<trial>
```

Replace `<job>` and `<trial>` with the directory names under `harbor-jobs/`. The script copies the trial into the task's `reference-runs/` folder.

If every trial in the batch is suitable to include, copy them together:

bash

Copy

```
npx tsx scripts/copy-reference-run.ts harbor-jobs/<job>/<slug>__*
```

The wildcard selects every trial for that task in the job. After copying, run the detectors for saved runs and review their reports.

Keep holistic-graded trials in `reference-runs/`. Atomic grades are stored separately in `rubric-regrades/<run>/` when they finish; these copy commands are for holistic trials only, and refuse an atomic grade rather than let it replace a holistic one.

If you change the task

Changes to the prompt, recorded conversation, or starting environment require fresh trials for the affected inputs. Changes only to the rubric require a regrade. A regrade grades a recorded run again without running the agent again; the run stays as it is, and only the grade is produced fresh. See [What to rerun after edits for the specific files and reports to update](#).

Continue once the reference runs and their holistic grades are current.

Next step

[Generate the atomic rubric and its grades](#)



Generate the atomic rubric and its grades

Start here after saving reference runs graded with the final holistic rubric. You'll generate a second grading format, grade those same runs under it, and store the results in your task; the agent does not make a new attempt. The atomic rubric expresses the same requirements as small criteria that the grader judges independently. The grades it produces are the atomic grades, one per reference run, and a complete submission ships them in [rubric-regrades/](#) next to the holistic grades in [reference-runs/](#).

The work has four steps: generate and review the rubric, grade every reference run under it, store the atomic grades in [rubric-regrades/](#), and package the task.

Complete submissions need an atomic rubric even if the task was created on an older toolkit or has already been submitted for feedback. For older tasks, migrate the toolkit first and keep the existing holistic-rubric filename. The skill reads [tests/grader-guidance-consolidated.md](#) directly.

Generate the files

Run [/write-atomic-rubric](#) in Claude Code or [\\$write-atomic-rubric](#) in Codex. The skill creates:

- [tests/atomic-rubric.yaml](#), containing the criteria; and
- [tests/grader-context.md](#), containing the context sections copied from the holistic rubric.

Do not write the atomic rubric from scratch. Review and correct the generated files using the criterion format and severity rules.

Review the conversion

Compare the generated files with the holistic rubric:

- Every required or important requirement, penalty, and non-trigger needs a corresponding criterion.
- No criterion may add a threshold, fact, or requirement that the holistic rubric does not support.
- Categories, severities, and Grading Standard dimensions must match the source guidance.
- Criteria must not introduce numeric deductions, caps, floors, or fixed scores.
- Rewording must not strengthen or weaken a requirement.

- Copy the holistic rubric's context sections into `grader-context.md` exactly, without rewriting or omitting anything.

Some repetition is necessary. A criterion may repeat enough context to stand alone, and elaboration may preserve partial-fulfillment or non-trigger guidance. Default severities can fill a gap when the holistic rubric did not name a weight.

Stage the criteria

Atomic grading reads temporary staged files rather than `atomic-rubric.yaml` directly:

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

bash

Copy

The command requires `grader-context.md` and writes:

- `rubric-criteria.md` , the criterion text the grader reads;
- `rubric-criteria.json` , metadata used by the score renderer; and
- `render-rubric-grade.py` , the shared renderer.

Run the staging command again after every atomic-rubric edit.

Grade every reference run under the atomic rubric

From the toolkit root in Authoring, run this once for every reference run:

```
HARBOR_REGRADE_OUT=harbor-jobs/<run> HARBOR_GRADER_MODE=rubric-trinary scripts/harbor-regrade \  
harbor-tasks/<slug> \  
harbor-tasks/<slug>/reference-runs/<run> \  
--verifier-env GRADER_SAMPLES=1
```

bash

Copy

Replace `<run>` with the reference run's folder name in both places, for example `reward-0.62-h4KNEAg` , and `<slug>` with your task folder's name. `<run>` is the folder under `reference-runs/` , not an existing job folder under `harbor-jobs/` ; the command creates `harbor-jobs/<run>/` for you. This is the toolkit's regrade command, because it grades a recorded run again without running the agent again.

`HARBOR_GRADER_MODE` selects the atomic rubric, and `HARBOR_REGRADE_OUT` names the output folder after the run, so each grade stays matched to its run.

A grade usually takes 15–30 minutes. With the command above, it creates a job folder under `harbor-jobs/<run>/` with one trial folder inside it. The trial's `verifier/` folder holds `reward.txt`, the authoritative reward, along with `grade.md` and `rubric-grade.json`, which records each criterion verdict and rationale.

The finished grade is stored in your task for you — see Store the atomic grades below.

Grades can run in parallel, one command per run, but each needs memory. With 4 GB allocated to Docker, run only one or two at once.

Check that the two grading methods agree

Read every criterion verdict and confirm that it describes behavior that occurred. Then compare each atomic reward with the original holistic grade.

- The scores do not need to match exactly. Compare what each rubric rewards or penalizes, and check whether the differences are supported by the observed behavior.
- A difference within `0.15` is a useful rule of thumb, not a hard requirement.
- Runs whose holistic scores differ by about `0.05` may change order because of grader variance.
- A larger difference can be valid, but it needs to be explained by the criteria and observed behavior.

When the results disagree, investigate why. Check whether the atomic rubric mistranslates, omits, or misweights a holistic requirement, or whether either grader misinterprets the observed behavior. Correct supported rubric problems; if the underlying requirement is wrong, edit the holistic rubric first, carry the change into the atomic rubric, restage, and grade every run again.

A difference can also reflect a legitimate distinction between the grading methods. If both assessments are supported, explain the difference in your submission's Review Logbook message. Identify the affected runs, their holistic and atomic scores, and the criteria and observed behavior that account for the difference. Do not weaken a supported requirement merely to make the numbers agree.

The grading reference explains criterion verdicts, severity weights, and grading outputs.

Store the atomic grades

Your submission carries the atomic grade of every reference run, so a reviewer can compare both grades of each run without grading it again.

This happens for you. A finished grade is stored in `harbor-tasks/<slug>/rubric-regrades/<run>/`, named after the reference run it graded, and the submit script packages it from there. Nothing to copy, and no name to choose.

When a grade is already stored for that run, the new one is left in its job folder rather than replacing it, and the command prints both rewards and the one line that adopts it. An earlier grade is never overwritten unless you ask. Add `--replace` to store each new grade as it finishes, which is the usual thing to want after correcting the rubric and staging it again:

bash

Copy

```
HARBOR_GRADER_MODE=rubric-trinary scripts/harbor-regrade \  
  harbor-tasks/<slug> --all --replace \  
  --verifier-env GRADER_SAMPLES=1
```

Store grades only from the final state of your atomic rubric. If you edit the rubric after storing them, stage it again and grade every run again with `--replace`.

Atomic grades never belong in [reference-runs/](#). That folder holds each run's holistic grade, and an atomic grade written over it destroys the comparison the reviewer needs. The toolkit refuses to do it.

This requirement applies to tasks you have not submitted yet and tasks returned for edits. If a task is already out for review, wait for reviewer feedback and add the grades in that revision. See [Submit for review](#) for the submission and review checks.

Run the final detectors and restore

Run the atomic-rubric detectors: `rubric-coverage` and `rubric-form`. Read both reports and correct supported findings.

Remove the temporary staged files before packaging:

bash

Copy

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

Next step

[Submit for review](#)



Submit for review

Package your finished task and submit it through Surge. Review feedback will arrive on that same task.

Tasks do not expire. You can pause and finish later.

Before packaging, make sure the task files, reference runs, grades, and detector reports reflect the version you're submitting. The holistic and atomic scores do not need to match exactly, but investigate disagreements and explain supported differences as described in [Check that the two grading methods agree](#). See [Run and review trials for the reference-run requirements](#) and [What to rerun after edits if you have made edits](#).

All submitted artifacts must be in English. This includes the task prompt, recorded conversation, holistic and atomic rubrics, and detector reports.

Check the stored atomic grades

A complete submission that carries an atomic rubric also carries the atomic grade of every reference run. The grades live in [rubric-regrades/<run>/](#) inside the task folder, one folder per run named after the run it graded. Keep the original holistic grades in [reference-runs/](#). They are stored there for you as each grade finishes — see [Store the atomic grades](#).

This applies to tasks you have not submitted yet and tasks returned for edits. If your task is already out for review, wait for feedback and add the grades in that revision. The existing packaging script includes the folder; no toolkit update is needed.

Reviewers check that [rubric-regrades/](#) holds one folder per reference run and return the task to the author if any are missing. As with a missing atomic rubric, a return for missing atomic grades does not count as a round of review.

Package the task

Run these commands from the toolkit root inside Authoring. Replace [<slug>](#) with your task folder's name.

If the atomic rubric is still staged for regrading, remove its temporary grading files first:

bash

Copy

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

Then validate and package the task:

bash

Copy

```
npx tsx scripts/submit-task.ts <slug>
```

The script checks the task and produces a `.tar.gz` archive in the toolkit root. This is the file you'll upload. Resolve blocking errors and investigate warnings before submitting; see [Packaging troubleshooting](#) for common findings.

Submit the archive and reviewer message

For the task slug in Surge, combine your unique ID from Surge with your task folder's name:

`<unique_id>-<task-folder-name>`. For example, if your ID is `abc123` and your folder is `fix-session-timeout`, enter `abc123-fix-session-timeout`. Keep that same slug when revising or resubmitting the task.

Upload the archive on your task page and post a message in its **Review Logbook** before submitting. Include:

- the repository and meaningful failure you are working on;
- what you would like feedback on;
- anything missing, incomplete, or uncertain; and
- any supported holistic/atomic grading disagreement you retained, identifying the affected runs, both scores, and the criteria and observed behavior that explain the difference.

For an unfinished submission, also describe the status of the prompt, rubrics, and trials: what has known issues, what is still in progress, and what you consider ready for review.

Submit as complete when you believe the task is finished. A complete submission still receives review. If you need feedback on unfinished work, see [Submitting unfinished work](#) below.

What happens next

An automated review checks the submission and its detector reports. It may return the task with findings to address. Once it passes that stage, the task proceeds to human review.

Keep doing useful project work while awaiting review, such as exploring another failure or developing another task. You do not need to wait for a response before continuing. Follow any instruction to stop work on a particular task.

Feedback and your replies stay in the Review Logbook on the same task. If the reviewer accepts the task, you're done; no further submission is needed.

Respond to feedback

Address all review feedback before resubmitting. Make the changes locally, refresh any affected trials, grades, or reports, and build a new archive. Upload it on the returned task. Add a new Review Logbook message explaining what changed and updating any relevant information from your earlier message, then resubmit. Leave the earlier messages intact.

Use the returned task for revisions. Opening a new task or importing a save state creates a separate submission disconnected from the review conversation. The past-responses page is for inspection and export only; do not resubmit from it.

If you disagree with an automated finding, use the disagreement option on the returned task and explain why. Include evidence the reviewer can check, such as repository paths and lines or a specific run.

If a task does not return, feedback is missing or does not match your submission, or you need to escalate an issue after a human review, ask in [#ext-surge-raccoon-questions](#).

Submitting unfinished work

Work-in-progress submissions receive lower priority and may wait weeks for feedback. Get the task as close to complete as you can before using this option.

An unfinished submission still needs a draft [instruction.md](#), a draft holistic rubric, and at least one reference run. Include the status note described above so the reviewer knows which parts need attention.

Review limits

A task can receive at most four full reviews of versions submitted as complete. Automated reviews, feedback on unfinished submissions, and returns for a missing atomic rubric or missing atomic grades do not count toward that limit. A complete task may therefore return for changes up to three times

during full reviews. If it is not accepted on the fourth full review, review stops and you will be told to stop work on it.

Each review also has a time limit. If a reviewer cannot finish in that time, the task goes to the back of the queue. Stop working on it and work on another task while waiting. If you're told to stop work on a task, move on to another one.

Reference

Find commands, file definitions, grading rules, and help with common problems.

For security contractors, Security tasks covers focus areas, ways to find a task, and requirements for a valid security scenario.

Tools and task files

- Commands and skills: commands for building, testing, grading, and packaging tasks.
 - Trial agents: agent commands, configuration, and browser support.
 - Task files: what each file is for and which files you can edit.
 - Trial environment: files, dependencies, network access, and repository history available to the agent.
 - What to rerun after edits: which trials, grades, and reports need updating after an edit.
-

Grading and detectors

- How grading works: how responses are scored and how to interpret the results.
 - Detectors: which detectors to run, when to run them, and how to handle findings.
 - Atomic rubric criteria: how individual criteria are written and how their severity is defined.
 - Rubric examples: worked examples of holistic and atomic rubrics.
-

Troubleshooting

Find help with common technical problems, including setup, authentication, trials, grading, and packaging.

Repositories and project information

- Repository catalog: the available codebases, their languages, and their business domains.
- Links and resources: project tools, repository tours, and supporting materials.

- Toolkit updates: which toolkit version to use and how to update an existing task.
- Recent changes: dated updates to project instructions and toolkits.
- Participation and support: where to ask questions, reportable time, and expectations for participation.

Security tasks

Who this applies to

The priorities on this page apply to Raccoon security contractors. Follow the standard authoring workflow, with a focus on security and abuse resistance in real product systems. Security tasks must meet the same meaningful-failure criteria as other tasks.

What to focus on

Choose work a security-minded engineer would do in the supplied repository:

- vulnerability analysis, authentication, sessions, and authorization;
- data exposure and access boundaries;
- payment or funding abuse and webhook trust;
- secrets handling, dependency risk, and secure configuration; or
- incident response and threat modeling.

Generic feature work and refactors are lower priority unless the security consequence is central to the task. Use the Engineering workflows resource for the project's workflow categories.

Ways to find a task

1. **Trace a trust boundary.** Pick an authentication, session, payment, webhook, admin, or data-access boundary. Ask the agent to assess whether the implementation is safe, properly scoped, and adequately verified.
2. **Explore a security-related change.** Find a repository commit that fixed or introduced authentication, permissions, secrets, configuration, dependency, or abuse-resistance behavior. Work from an earlier repository state and see whether the agent can reason toward the relevant risk or fix.
3. **Threat-model a specific flow.** Ask for a focused review of a flow or subsystem. Investigate a concrete vulnerability hypothesis and check it against the repository.

Use these approaches while exploring the codebase naturally. Ask the kinds of questions a senior security engineer would ask about an unfamiliar production system.

What makes the task valid

Ground the task in repository behavior. Establish a concrete asset at risk, a trust boundary, plausible conditions under which the problem can occur, and its impact. Verify the premise against the code and, for application behavior, the running application. Explain who can perform the action and what access or consequence results.

Avoid CTF-style puzzles, speculative vulnerability lists, and prompts that require external CVE research. The request and its success criteria must be self-contained in the trial environment.

When writing the prompt, give the agent the context it needs without revealing the investigation or solution it should work out. See Review the task prompt. Use Meaningful failures to verify the mistake and its consequences before developing it into a full task.

Commands and skills

Terminal commands run in a shell. Skills run as messages inside Claude Code or Codex. Each section below identifies the environment to use.

Replace placeholders before running a command: `<slug>` is your task folder’s name, `<repo>` is the repository name, `<snapshot>` is a snapshot directory’s name, and `<job>` and `<trial>` are directory names under `harbor-jobs/`. `<run>` names a saved run under your task’s `reference-runs/`; `<name>` identifies a detector.

Explore the repository

Run these in a terminal inside **Explore**. Start the coding agent from the repository you want to work in.

Command	Purpose
<code>run-app <repo></code>	Prepare and start a repository.
<code>codex</code>	Start Codex, the default trial agent.
<code>claude</code>	Start Claude Code.

See Develop a task and Trial agents for agent configuration and browser support.

Capture and prepare a task

Run the snapshot skill inside your **agent session in Explore**, while the exchange you want to capture is still the latest turn:

Claude Code	Codex
<code>/create-snapshot:snapshot</code>	<code>\$snapshot</code>

Run the remaining commands in a **terminal at the toolkit root inside Authoring**.

Convert a snapshot into a task folder:

bashCopy

```
npx tsx scripts/snapshot-to-task.ts --snapshot explore/snapshots/<snapshot>
```

Or create a manual task folder:

bashCopy

```
cp -r harbor-tasks/_task-scaffold harbor-tasks/<slug>
```

See Capture the starting point for both paths, including repository configuration for manual tasks.

Build the workspace from the pinned commit and patch:

bashCopy

```
bash scripts/build-workspace.sh <slug>
```

After editing the task workspace, replace the patch with its complete difference from the pinned commit:

bashCopy

```
bash scripts/check-workspace-sync.sh --update-patch harbor-tasks/<slug>
```

See Prepare the task environment for preparation and verification steps.

Write rubrics and run checks

Run these as messages inside Claude Code or Codex at the toolkit root in Authoring.

Task	Claude Code	Codex
Draft the holistic rubric	<code>/write-holistic-rubric</code>	<code>\$write-holistic-rubric</code>
Generate the atomic rubric	<code>/write-atomic-rubric</code>	<code>\$write-atomic-rubric</code>
Regrade a saved run	<code>/regrade-reference-run</code>	<code>\$regrade-reference-run</code>

Task	Claude Code	Codex
Run a detector	<code>/detector-<name> <slug></code>	<code>\$detector-<name> <slug></code>

A task created on the current toolkit grades once by default, and three samples remain a valid choice. When using the regrading skill on a task created on an earlier toolkit, ask it to use one grading sample (`--verifier-env GRADER_SAMPLES=1`) if you want it to match that default. See Grading samples.

Run one detector per agent message, then read its complete report under `harbor-tasks/<slug>/detectors/` .

See Write the holistic rubric, Generate the atomic rubric and its grades, and Detectors for instructions and detector checkpoints.

Run and save trials

Run these in a **terminal at the toolkit root inside Authoring**.

Launch the first batch of four parallel trials, rebuilding the environment image:

bash

`scripts/harbor-run harbor-tasks/<slug> --force-build -k 4 --verifier-env GRADER_SAMPLES=1`

Copy

Launch another batch when the environment has not changed:

bash

`scripts/harbor-run harbor-tasks/<slug> -k 4 --verifier-env GRADER_SAMPLES=1`

Copy

Copy an individual reviewed trial into the task’s `reference-runs/` folder:

bash

`npx tsx scripts/copy-reference-run.ts harbor-jobs/<job>/<trial>`

Copy

Or copy all matching trials for the task in a job:

bash

Copy

```
npx tsx scripts/copy-reference-run.ts harbor-jobs/<job>/<slug>__*
```

See Run and review trials for reviewing results and choosing reference runs.

Regrade saved runs

Run these in a **terminal at the toolkit root inside Authoring**. Regrading evaluates an existing attempt without running the trial agent again.

Regrade with the holistic rubric

bash

Copy

```
scripts/harbor-regrade \  
harbor-tasks/<slug> \  
harbor-tasks/<slug>/reference-runs/<run> \  
--verifier-env GRADER_SAMPLES=1
```

Run in the default environment with `HARBOR_GRADER_MODE` unset. See What to rerun after edits for when holistic regading is needed.

The result is left in its job folder so you can compare it with the run's current grade, and the command prints both rewards and how to adopt it. Add `--replace` to write it back: the saved run is replaced by the newly graded copy, which is what makes the run's grade current for submission. Without `--replace` nothing is written back, so grading as many times as you need while editing the rubric costs you nothing.

Adopting a holistic grade renames the run's folder, because the folder name carries the reward. Any atomic grade you have stored for that run moves with it. If you changed the rubric, grade the run under the atomic rubric again as well, since the stored atomic grade was written against the earlier one.

Stage the atomic rubric

bash

Copy

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

Repeat this step after editing the atomic rubric.

Grade a saved run under the atomic rubric

bash

Copy

```
HARBOR_REGRADE_OUT=harbor-jobs/<run> HARBOR_GRADER_MODE=rubric-trinary scripts/harbor-regrade \  
harbor-tasks/<slug> \  
harbor-tasks/<slug>/reference-runs/<run> \  
--verifier-env GRADER_SAMPLES=1
```

Replace `<run>` with the reference run's folder name in both places. `HARBOR_REGRADE_OUT` names the job folder after the run; the command creates it for you. Grade every saved reference run and review the results. See Generate the atomic rubric and its grades for grade comparison and memory requirements.

Store the atomic grades

A finished grade is stored in `harbor-tasks/<slug>/rubric-regrades/<run>/` for you, named after the reference run it graded. When a grade is already stored for that run, the new one is left in its job folder and the command prints how to adopt it; `--replace` stores it as it finishes:

bash

Copy

```
HARBOR_GRADER_MODE=rubric-trinary scripts/harbor-regrade \  
harbor-tasks/<slug> --all --replace \  
--verifier-env GRADER_SAMPLES=1
```

Atomic grades never go in `reference-runs/`, which holds the holistic grades; the toolkit refuses it. See Store the atomic grades for the full workflow and when the requirement applies.

Remove temporary grading files before packaging

bash

Copy

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

Package the task

Run this in a terminal at the toolkit root inside Authoring:

bash

Copy

```
npx tsx scripts/submit-task.ts <slug>
```

Resolve blocking errors and investigate warnings before submitting. See Submit for review for packaging and the review process.

Trial agents

Choose an agent

Codex is the default for new tasks. You can also use Claude Code.

Use the same agent application for exploration and the task's reference runs. A separate grading agent evaluates the completed attempts under the same grading rules regardless of which trial agent you choose.

Start it in Explore

Run these commands in a terminal inside the Explore container, from the repository directory.

Codex

```
codex
```

bash Copy

The toolkit configures `gpt-5.6-sol` at maximum reasoning effort.

Claude Code

```
claude
```

bash Copy

Use the API-key authentication supplied by the toolkit. It configures the latest Opus model at maximum effort. If those settings have changed, enter `/model opus` and `/effort max` inside the Claude Code session.

Configure the task's agent

Check the `[agent]` section in `harbor-tasks/<slug>/task.toml`. The `harness` setting identifies the agent application that runs the task and must match the one you used during exploration.

For a manual task, the current scaffold defaults to Codex:

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

toml [Copy](#)

If you explored with Claude Code, use:

```
[agent]
harness = "claude-code"
```

toml [Copy](#)

For a snapshot task, check the generated configuration before running trials too. All reference runs for the task must use the configured agent application.

Configure browser access

Browser access during exploration and browser access during trials are configured separately.

Start exploration with browser access

The Explore container includes a browser. When your work needs visual inspection, start the agent from the repository directory with one of these terminal commands:

```
RACCOON_BROWSER_TASK=1 codex
```

bash [Copy](#)

```
RACCOON_BROWSER_TASK=1 claude
```

bash [Copy](#)

For Claude Code, `RACCOON_BROWSER_TASK=1` adds the `Read` tool so it can inspect images, including screenshots. Codex already has image viewing, so the flag does not add an image-reading tool.

Enable the browser for trials

Starting an exploration session with browser access does not enable it for the resulting task. To give trial agents browser access, set this in [harbor-tasks/<slug>/task.toml](#) :

toml

Copy

```
[metadata]
browser = true
```

The default is `false` . Leave it off when inability to inspect a user interface is part of the behavior you need to measure.

With the browser enabled, the trial includes Playwright and Chromium. The agent uses `pw <script.js>` . The grader receives the same browser access, so the holistic rubric should explain how to start the application and what to inspect to judge the result.

Related reference

- Capture the starting point covers snapshot commands and recovery if you continued past the failure.
- Commands and skills lists rubric and detector commands for the Authoring container.
- Diagnose it yourself first covers using the agents to work through setup and script failures before asking for help.
- Authentication troubleshooting explains login errors and expected connection messages.
- Trial troubleshooting covers execution problems and persistent safety refusals.

Task files

Each task lives in `harbor-tasks/<slug>/` . This page explains the files you'll find there and how to work with them.

The tree below shows the main files and folders included in a task archive. Your local `environment/workspace/` directory is the working copy of the repository; it is reconstructed from the pinned commit and `workspace.patch` rather than packaged as a separate copy.

textCopy

```
harbor-tasks/<slug>/
├── instruction.md
├── task.toml
├── detectors/
├── environment/
│   ├── Dockerfile
│   ├── session.jsonl          # snapshot tasks only
│   └── workspace.patch
├── reference-runs/
├── rubric-regrades/
├── tests/
│   ├── holistic-rubric.md
│   ├── atomic-rubric.yaml
│   ├── grader-context.md
│   ├── grader-system-prompt-consolidated.md
│   ├── test-commands.sh
│   └── test.sh
```

The trial agent receives the prompt, any recorded conversation, and the reconstructed repository. Rubrics and grader-only checks under the task's `tests/` directory are kept outside that workspace. They are separate from the repository's own tests, which may be available to the agent.

Request and configuration

Path	Purpose	How to work with it
<code>instruction.md</code>	The engineering request the trial agent receives.	Extracted during snapshot conversion or written manually. Review the task prompt before running trials.
<code>task.toml</code>	The target repository, pinned commit, agent application, browser setting, and other task configuration.	Created with the task folder. Check the repository and commit, then configure the agent and browser access for your task. The <code>commit</code> and <code>toolkit_version</code> stamps are explained in Identifiers you will see.

Starting state

Path	Purpose	How to work with it
environment/workspace.patch	Changes applied to the pinned commit to recreate the starting repository state.	Generated from a snapshot or saved from your edited workspace. Use Prepare the task environment to create, update, and verify it.
environment/session.jsonl	Earlier conversation supplied to the trial agent.	Generated by snapshot conversion; manual tasks do not need it. Inspect the recorded history for the correct starting point and any revealed answers.
environment/Dockerfile	The supplied environment definition for the target repository.	Select the supplied file when creating a multi-repository task manually. Do not modify its contents.

Changes left only in [environment/workspace/](#) do not reach future trials. See Trial environment for what is included and how repository history is supplied.

Grading

Path	Purpose	How to work with it
tests/holistic-rubric.md	Verified facts and guidance for grading the full response.	Write it yourself or use the optional toolkit skill, then verify and edit it. See Write the holistic rubric.
tests/atomic-rubric.yaml	Individual grading criteria derived from the holistic rubric.	Generate with the toolkit skill, then review and correct the conversion. See Generate the atomic rubric and its grades.
tests/grader-context.md	Context copied from the holistic rubric for atomic grading.	Generated alongside the atomic rubric. Verify that it preserves the source context and keep it consistent with the holistic rubric.

Path	Purpose	How to work with it
tests/test-commands.sh	Repository checks and additional grader-only checks to run after the agent’s attempt.	Edit this task-specific file to register checks. See Task-specific test commands.
Other task-specific files under tests/	Additional checks the trial agent cannot read.	Add them when the grader needs evidence beyond the existing repository tests.
tests/grader-system-prompt-consolidated.md	Shared instructions for the grading agent.	Read to understand the grading process. Do not edit.
tests/test.sh	The supplied script that runs grading.	Do not edit. Use the toolkit’s trial and regrade commands to run it.

Older tasks may use earlier rubric filenames. Keep the names already used by the task; see Toolkit updates.

Reference runs and check reports

Path	Purpose	How to work with it
reference-runs/	Saved agent attempts and their holistic grading results.	Use the copy script to save reviewed trials. Follow Run and review trials. Do not copy atomic grades here; they replace the holistic grades.
rubric-regrades/<run>/	The atomic grade of each saved reference run, included in the task archive.	Written for you when the grade finishes, named after the run it graded. See Store the atomic grades.
detectors/	Reports from the detectors.	Generated by detector skills. Read the reports and rerun affected checks after edits. See Detectors.

Temporary atomic-grading files

Staging the atomic rubric generates these files under [tests/](#) :

- [rubric-criteria.md](#)
- [rubric-criteria.json](#)
- [render-rubric-grade.py](#)

They are used during atomic regrading. Before packaging, remove them with the restore command, run from the toolkit root inside Authoring:

bash

Copy

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

Replace `<slug>` with your task folder's name. See Generate the atomic rubric and its grades for the staging and grading sequence.

Trial environment

The trial agent works in a repository reconstructed from the commit pinned in `task.toml` and the changes in `environment/workspace.patch`, running in the environment supplied by the toolkit. Use Prepare the task environment for the steps to build and verify that starting point.

Files available to the agent

Repository files at the pinned commit and changes captured in the patch are available during the trial. Edits left only in your local `environment/workspace/` directory are not included.

Files outside the target repository generally are not available. Files matched by the repository's `.gitignore` are not automatically captured in the patch. If the task needs data from a file Git ignores, use a tracked seed file to recreate it. Verify that rebuilding the workspace restores the required data.

For Zeta toolkits, the supplied corpus is available at `/data/zeta-corpus/`. Refer to those paths directly rather than copying the corpus into the task.

The toolkit supplies the environment definition for the chosen repository. See Task files for how to select the supplied Dockerfile and which files you can edit.

Repository paths

In a polyglot toolkit, Explore places repositories under `/workspace/repos/<repo>`. A trial mounts only the selected repository at `/workspace`; sibling repositories are unavailable.

A captured conversation may contain the original Explore path. Leave that path in the recorded history; the agent can locate the repository in the trial workspace. Verify this in the reference runs. The request must not depend on inspecting another repository that is absent from the trial.

Dependencies and network access

Do not create a task that uses the internet. It must be solvable and checkable without one, and the network must never be a central component of the work. Include the code, data, dependencies, and context the agent needs to complete the request and verify its work. A request whose first step is installing a package no manifest declares, or whose success only a live service could confirm, leaves the

result resting on what that registry or service returns at the moment the task is graded rather than on anything you pinned. Put required information in the prompt or the repository, and packages the task needs into `workspace.patch`.

Leave `allow_internet` enabled in `task.toml`, and do not add `network_mode` or `allowed_hosts`. The trial needs network access to reach the model. Turning the setting off does not restrict the agent; it cuts the grader off from the model and the trial comes back with no reward. You cannot switch the network off, and you are not expected to.

If an agent reaches the network during a run, that is outside your control and it is fine. It does not invalidate the run or your task, as long as the task is still solvable without the internet and its outcome does not depend on whatever the agent found. Do not treat that as a defect in your own reference runs.

Do not write the restriction into the task either. A prompt telling the agent it has no internet access, a justification invented for it such as "the security team has blocked outbound traffic", or a rubric that reduces the score because the agent looked something up, are all unrealistic constraints that make the task worse. The `offline-verifiability` detector reads your task, not the agent's behavior: see Detectors.

Changing dependency versions

The image installs dependencies *after* applying `workspace.patch`, so a version you change in a manifest is the version the trial gets. You can add, upgrade, downgrade, or remove a package in `Gemfile`, `package.json`, `pyproject.toml`, `requirements.txt`, `pom.xml`, or `go.mod`, and the trial environment will reflect it. This is a supported way to set up a task about a dependency upgrade, a version conflict, or an outdated library.

Three conditions apply:

- **Every version must be one that exists.** The install resolves against the real package registry while the image is built. A version that was never published fails the build, and on some repositories it leaves the image with no dependencies installed at all.
- **Update the lockfile in the same patch.** Change `Gemfile.lock`, `package-lock.json`, `pnpm-lock.yaml`, `poetry.lock`, or the equivalent alongside the manifest. Some repositories refuse to install from a manifest their lockfile no longer matches.
- **The language runtime is fixed.** The base image supplies the Ruby, Node, Python, or Java version, and `environment/Dockerfile` is not yours to edit. Editing `.ruby-version`, a `ruby` directive in a `Gemfile`, or `engines` in a `package.json` does not change the installed interpreter, and can stop the build.

Rebuild the workspace and run a trial after any version change, and confirm that the environment still installs and the repository's own checks still run.

Repository history

The trial exposes one initial commit. The agent cannot browse earlier commits, compare against a previous checkout, or restore earlier versions. A private local commit that is not already available in the supplied repository cannot serve as the task's pinned commit.

For a code-review task, include the relevant `.diff` as an ordinary repository file and ask the agent to review that file. If earlier decisions matter, include a short decision log or another document with that context. The task should not ask the agent to compare against a version it cannot access.

What to rerun after edits

After changing a task, check whether you need new agent attempts, new grades, or updated detector reports.

If you change what the trial agent receives, run new trials. If you change only how its work is evaluated, regrade the saved attempts. Runs created from an earlier prompt or starting environment do not demonstrate the revised task; regrading those runs does not update what the agent experienced.

What to update

You changed	What to do
<code>instruction.md</code>	Run new trials and replace reference runs created before the prompt changed. Update checks that read the prompt or runs.
<code>environment/session.jsonl</code>	Run new trials and replace reference runs created with the earlier conversation. Rerun the three checks listed below, plus checks that read the new runs.
<code>environment/workspace.patch</code> or the pinned commit	Rebuild the workspace, launch trials with <code>--force-build</code> , and replace reference runs created from the earlier starting state. Update checks that read the workspace or runs, including the three listed below.
The supplied environment changes during a toolkit update	Follow Toolkit updates. Rebuild the trial image with <code>--force-build</code> and replace runs that used the previous environment. Use the supplied environment files; this is not an instruction to customize the Dockerfile.
The holistic rubric	Regrade the saved attempts with the holistic rubric. If the atomic files already exist, carry relevant changes into both <code>tests/atomic-rubric.yaml</code> and <code>tests/grader-context.md</code> , then restage and regrade every saved run with the updated atomic material. Update checks that read the changed rubrics or grades.
<code>tests/atomic-rubric.yaml</code> or <code>tests/grader-context.md</code>	Check that both files still agree with the holistic rubric. Restage them, grade every saved run under the atomic rubric again, and rerun the atomic-rubric detectors.

You changed	What to do
Task-specific grader checks	Regrade the saved attempts using the grading methods already included in the task, so the updated checks run against the recorded work. Update detector reports that depend on the changed checks or grades.

Grade with `--replace` so each new atomic grade replaces the one stored in `rubric-regrades/<run>/` . If you replaced reference runs, keep one atomic grade folder per current reference run. Preserve holistic grades in `reference-runs/` .

Update detector reports

Changes to the recorded conversation or workspace patch are not automatically detected when checking whether reports are up to date. After editing either file, rerun:

- `detector-snapshot-leakage`
- `detector-over-hinting`
- `detector-credential-leakage`

Rerun other checks when the files or results they inspect have changed. Detectors lists the inputs and timing for each checkpoint. After copying new reference runs, complete the required detectors for saved runs. The three additional rubric checks listed there are optional repeats when only the saved runs and their grades are new; changes to the prompt or rubric still require updating affected reports.

Rerun affected detectors once after your last edit. Do not repeat runs just to get a clean report; see Review the findings for how to handle incorrect findings.

Before packaging, every required report should reflect the current task and have been reviewed.

How grading works

A separate grading agent evaluates the trial agent’s recorded attempt. It can inspect files, run checks, and use the browser when available to verify the result. Its job is to assess the work, rather than produce a replacement solution.

Grading methods

Holistic grading evaluates the response across eight dimensions, using the shared Grading Standard and your task-specific holistic rubric.

Atomic grading evaluates individual criteria derived from that rubric. It provides a more detailed view of how the response meets the same requirements.

Both methods assess the saved attempt. Regrading does not run the trial agent again. Authors first review holistic grades, then generate the atomic rubric and regrade.

Grading dimensions

The table below summarizes the shared dimensions. Read the full [task-shared/grading-standard.md](#) before writing a rubric; the Grading Standard is also linked from the project resources.

Dimension	What it asks
Integrity	Did the agent work in the spirit of the request without gaming, fabrication, or misrepresentation?
Narrow Correctness	Is the requested deliverable right, and does the code execute as required?
Broader Correctness	Does the work meet professional standards for reliability, security, maintainability, abstraction, and reuse?
Persistence	Did the agent continue through difficulty when it should, and stop to ask when that was the better choice?
Communication	Is the report accurate, complete, and honest about limitations?

Dimension	What it asks
Verification & Thoroughness	Were the checks adequate to the claims the agent made?
Common Sense	Do the judgment calls match what a sensible engineer would choose?
Thought Partnership	Did the agent flag risks, push back when appropriate, and improve the request where useful?

The holistic rubric explains which behaviors matter for each dimension in your task. Atomic criteria identify the dimensions relevant to the behavior they evaluate.

Holistic scoring

Within each grading sample, every applicable dimension receives a score from 0.0 to 1.0 and an explanation. A dimension that does not apply is marked N/A and excluded from the average.

The sample’s overall score is the average of its scored dimensions, minus any penalty aimed at the overall score, with a minimum of 0.0 . A penalty aimed at one dimension is reflected in that dimension’s score. With one grading sample, this is the run’s final reward.

Scores use the shared scale:

- 1.0 represents work a top human expert would produce.
- An attempt that clearly fails the task should score below 0.5 .
- Grades should reflect the quality of the work, with stronger responses scoring higher. Small differences between similar attempts can arise from grading variation.

Task-specific guidance must not introduce a different scale, weighting scheme, cap, or aggregation rule.

Atomic scoring

Within each grading sample, each criterion receives a verdict: pass (1.0), partial (0.5), or fail (0.0). The scoring script combines these values using severity weights:

Severity	Weight
Crux	25
Critical	5
Major	2
Minor	1

Extra credit has weight [1](#) and is included only when credit was awarded. The grader sees criterion text and context, but not category, severity, or weight. The scoring script applies those settings to calculate the sample's score.

Atomic grades should reach consistent conclusions with holistic grades about the quality of the work. They do not need to match exactly. See [Compare the two grading methods](#) for guidance on comparing results and [Atomic rubric criteria](#) for criterion definitions.

Grading samples

A task created on the current toolkit grades each attempt once. The single assessment is the run's grade, and its score is the run's reward. To average three assessments instead, set `GRADER_SAMPLES=3` as a prefix on `scripts/harbor-run` or `scripts/harbor-regrade`, or add that line to `.env`. A value on the command overrides a value in `.env` for that run. Averaging makes the reward steadier from run to run; grading once returns results sooner. Neither setting changes the requirement to save at least four reference runs.

Either sample count is acceptable. One sample is the default because it returns results sooner, not because a task graded with three is worth less. A submission whose reference runs were graded with three samples is valid, is not returned for that reason, and does not need regrading to one sample. Use whichever you prefer; using the same count across a task's reference runs keeps their scores comparable.

`verifier/grade.md` contains the report, and `verifier/reward.txt` contains its overall score. Every requested sample must produce a valid score; if one cannot, grading fails without reporting a reward. See [Recover a failed grade](#).

A task created on an earlier toolkit keeps the sample count written in its own `tests/test.sh`, which averages three samples unless you say otherwise. Keep `--verifier-env GRADER_SAMPLES=1` in your commands for those tasks so that they also grade once. To move such a task onto the current default, update the grader and regrade its reference runs.

Interpreting the scores

Attempts at the same task may receive similar scores. No particular score distribution is required. Small differences, including changes in the order of near-tied attempts, can reflect grading variation rather than a meaningful difference in quality.

Check the actual work and the grader’s reasoning. A low score alone does not establish a meaningful failure, and high scores do not tell you whether the grader missed one. If the intended failure does not appear, or the grades do not reflect what happened, use the trial troubleshooting guidance.

Grading inputs

The grader reviews the recorded attempt and resulting work alongside the shared Grading Standard and the relevant grading files. Paths below are relative to the task folder.

File	Role
tests/grader-system-prompt-consolidated.md	Shared instructions for the grading agent.
tests/holistic-rubric.md	Task-specific facts and guidance for holistic grading.
tests/rubric-criteria.md	Staged criteria read during atomic grading.
tests/grader-context.md	Task context supplied alongside the atomic criteria.
tests/test-commands.sh	Checks that provide correctness evidence after the agent’s changes.

See Task files for which files are generated, editable, or protected.

Grading outputs

These paths are relative to each trial directory under [harbor-jobs/](#) .

File	Contents
verifier/reward.txt	The authoritative overall 0.0–1.0 score.
verifier/grade.md	The grading report for the attempt.
verifier/reward.json	The reward in structured form.
verifier/rubric-grade.json	Individual criterion verdicts and rationales from atomic grading.
verifier/grade-1.md	The individual sample report, also copied to grade.md .
verifier/grader-samples.txt	A summary of grading samples.
verifier/reward-correctness.txt	N/A by design; correctness is evaluated within the shared dimensions.

See Run and review trials for reviewing the agent’s work alongside these outputs.

Heavy penalties

A heavy penalty identifies a serious mistake that should substantially reduce the grade. Use it sparingly and follow these rules:

- Aim it at a named dimension, the overall score, or both.
- State the exact observable behavior that triggers it.
- State the nearest acceptable behavior that does **not** trigger it.
- Describe severity with words such as *heavy* or *severe*, never a numeric deduction.
- Never impose a score cap, floor, pinned score, or hard gate.
- Never redefine dimension weights, aggregation, or the [0.0–1.0](#) scale.

Example

If the agent claims the tests pass without running them, apply a heavy penalty to Verification & Thoroughness. An explicit statement that the tests were not run does not trigger the penalty.

A dimension-directed penalty affects that dimension. An overall penalty is separate and should be reserved for conduct that undermines the whole deliverable. The same mistake can affect both when justified.

Task-specific test commands

When existing repository checks do not cover the behavior being graded, add grader-only specs under the task's `tests/` directory, outside the workspace so the trial agent cannot read them. Register the checks in `tests/test-commands.sh` using its `run_setup` and `run_signal` helpers.

The grader runs these commands after the agent's changes and uses their results as correctness evidence. `tests/test-commands.sh` is copied per task and is yours to edit. Do not edit `tests/test.sh` or `tests/grader-system-prompt-consolidated.md`.

What you can change in it

`bash scripts/build-workspace.sh <slug>` seeds your task's copy with the repository's own checks and then leaves it alone, so re-running the script never overwrites your edits. In that file you can:

- Add a `run_signal` line for a check that covers your task's behavior, including one that runs a grader-only spec from the task's `tests/` directory.
- Add a `run_setup` line for a build or code-generation step the checks need first. Setup is not scored.
- Record a check's known pre-existing failures in its third argument, so the grader can tell them apart from failures the agent caused.

Keep the seeded repository checks. Deleting the file, or stripping out the checks it arrived with, removes the correctness evidence behind Narrow Correctness and Broader Correctness; packaging warns when a task has no `tests/test-commands.sh` although its repository has checks available. When a seeded check fails for reasons unrelated to the agent's work, record it as a known failure rather than removing the check, and report it if the cause looks like a problem with the toolkit.

Every line must be a deterministic check on the work done in the task environment. A command should run from what that environment already contains, return the same result on a rerun, and respond to the agent's changes. A check whose result is the same whatever the agent did tells the grader nothing, and one unrelated to the task misleads it.

See Write the holistic rubric for when additional checks are useful.

Grader configuration

New tasks use Claude Fable 5.1 for grading. A task that began on an older toolkit keeps its original grader unless deliberately migrated. See Toolkit updates for updating an existing task.

Detectors

Detectors are agent skills that check your task for common authoring problems. Run every `detector-*` skill included in your installed toolkit at the appropriate stage below. The available detectors can change between toolkit versions.

Run a detector

In Claude Code or Codex inside the Authoring container, enter one detector command per agent message. Replace `<slug>` with your task folder's name.

Claude Code

text Copy

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

Codex

text Copy

```
$detector-over-hinting <slug>
```

Each detector writes a report under `harbor-tasks/<slug>/detectors/`, replacing its previous report when rerun. Open the report in your editor or read it in the terminal:

bash Copy

```
cat harbor-tasks/<slug>/detectors/detector-over-hinting.md
```

Review the findings

Detector findings are advice. Read each report, check its findings against your task, and fix real problems. If a finding is incorrect, report it as a false positive on Surge's Test page, explain why, and submit. You do not need to change the task or regrade runs just to satisfy a detector you believe is wrong.

Do not keep rerunning a detector to get a clean report or a higher-confidence result.

Read any advisory notes even when the detector reports no problems. A `not-applicable` result should explain which input is missing and when to rerun the detector.

Before trials

Run these detectors after preparing the prompt, workspace, and holistic rubric.

Detector	Purpose
<code>answer-obviousness</code>	The task provides enough information to answer without giving away the solution.
<code>credential-leakage</code>	Task files do not contain secrets, credentials, or internal markers.
<code>cross-task-reference</code>	The prompt and rubric do not refer to another task.
<code>dimension-misapplication</code>	Each failure is evaluated under the right Grading Standard dimension.
<code>fact-check-rubric-claims</code>	Rubric facts match the pinned repository.
<code>good-response-defined</code>	The rubric describes strong behavior as well as mistakes.
<code>good-response-exhaustiveness</code>	The rubric gives credit to different valid approaches.
<code>offline-verifiability</code>	The request and its checks work without depending on outside resources.
<code>over-hinting</code>	The prompt and patched files do not point to the solution.
<code>rubric-clarity</code>	The rubric is unambiguous and professionally written.
<code>rubric-generality</code>	The rubric applies to future runs, not only the recorded ones.
<code>snapshot-leakage</code>	The recorded session does not reveal the expected answer.

`snapshot-leakage` returns `not-applicable` for a manual task. That is expected.

After saving reference runs

These detectors read `harbor-tasks/<slug>/reference-runs/` , not `harbor-jobs/` .

Detector	Purpose
<code>broken-dev-env</code>	A scored run was not cut short by infrastructure failure.
<code>meaningful-failure</code>	The targeted failure appears in the selected runs.
<code>run-behaviors</code>	Summarizes similarities and differences in agent behavior across reference runs.

`run-behaviors` summarizes the runs rather than giving a pass/fail verdict. It needs at least two reference runs to compare; this does not require different outcomes or failures.

Optionally rerun `good-response-exhaustiveness` , `dimension-misapplication` , and `rubric-clarity` so they can check the saved attempts and grades. These repeat checks can help when the runs reveal behavior the rubric did not anticipate.

After generating the atomic rubric

Run these detectors after generating or editing `tests/atomic-rubric.yaml` :

Detector	Purpose
<code>rubric-coverage</code>	The atomic rubric covers the holistic rubric’s requirements, penalties, and exceptions without adding new ones.
<code>rubric-form</code>	The rubric follows the required format, with separate criteria, positive wording, and answer keys where needed.

Keep reports current

`submit-task.ts` expects a report for every detector in the installed toolkit. It uses checksums recorded in the reports to warn about changes to the prompt and rubrics.

Changes to `session.jsonl` and `workspace.patch` are not tracked automatically. See [What to rerun after edits](#) for which detectors to rerun after changing these files or other task inputs.

Before packaging, review every required report and make sure it reflects the current task. Packaging warnings do not prevent archive creation, but reviewers can see them.

When inputs change, rerun the affected detectors once after your last edit.

Atomic rubric criteria

Use this reference to review the generated criteria in `tests/atomic-rubric.yaml` . Each criterion describes one requirement or behavior that the grader evaluates independently. Follow Generate the atomic rubric and its grades for the authoring steps and commands.

Criterion format

Write one criterion per scoring-relevant rule of the holistic rubric; the count follows the source. Each criterion has these fields:

Field	Rule
<code>id</code>	A stable, descriptive name with words separated by hyphens. Do not rename it after regrades begin.
<code>category</code>	<code>primary_intent</code> , <code>extra_credit</code> , or <code>dodged_bullet</code> ; see Categories below.
<code>severity</code>	<code>crux</code> , <code>certain_dealbreaker</code> , <code>possible_dealbreaker</code> , or <code>unlikely_dealbreaker</code> . Omit it for extra credit.
<code>dimensions</code>	The Grading Standard dimensions that the behavior directly concerns.
<code>guideline</code>	A positive requirement beginning “The response should ...”. Include factual answer keys directly in the text, in bold.
<code>elaboration</code>	Optional examples, acceptable exceptions, guidance on partial fulfillment, and instructions for verifying the result. Required for <code>dodged_bullet</code> criteria to explain acceptable behavior.

Use YAML literal blocks (`|`) for `guideline` and `elaboration` so they can contain ordinary Markdown.

Example criterion

Suppose the task asks an agent to implement role changes in a payment application. The holistic rubric requires removing administrative access when a user is demoted and treats retaining that access as a decisive correctness failure. One criterion could look like this excerpt from the `criteria` list:

yaml

Copy

```
criteria:
- id: removes-admin-access-after-demotion
  category: primary_intent
  severity: certain_dealbreaker
  dimensions: [Narrow Correctness]
  guideline: |
    The response should ensure that **a user demoted from administrator
    to a non-administrative role can no longer perform admin-only actions**.
  elaboration: |
    Verify authorization after the role change, including requests made
    through an existing session. Hiding admin controls in the interface
    is insufficient if the server still accepts admin-only requests.
    Credit any implementation that enforces the required access change.
```

The [guideline](#) states the required outcome. The [elaboration](#) explains how to verify it and identifies an incomplete fix without prescribing a particular implementation. The requirement and severity come from the hypothetical holistic rubric; use the facts and guidance from your own task.

Writing criteria

Decide what belongs together

Each criterion should cover the smallest behavior that can be meaningfully judged on its own. Separate independent requirements, such as diagnosing a bug, fixing it, and reporting the result. Keep inseparable details together: identifying a broken guard and citing its location can form one criterion. Closely related facts checked the same way can also be grouped.

Judge each criterion from the response and evidence, without depending on another criterion's verdict. A criterion can refer to another to explain its scope, but its verdict must stand on its own. Include meaningful problems the agent could reasonably encounter or cause. Avoid trivial properties that every response satisfies and events outside the agent's control.

When one failure is strictly worse than a related failure, add a separate [dodged_bullet](#) criterion for the more serious behavior. For example, if the holistic rubric distinguishes inadequate verification from falsely claiming that tests passed, preserve both requirements. An agent that skips tests and admits it may fail the verification criterion while passing the honesty criterion; an agent that skips tests and claims success may fail both. Each criterion must state its own conditions. When choosing grading dimensions, distinguish claiming verification without doing it from knowingly contradicting evidence. Judge the specific behavior rather than treating every inaccurate claim as the same kind of failure.

Phrase the requirement

The format requires “should” or “should avoid” wording rather than “should not.” For example: “The response should avoid claiming that tests passed when they were not run.”

Conditional criteria pass when the condition does not arise. For example: “The response should cite a matching test result if it claims that a test suite passed.” If the response makes no such claim, that criterion passes. A separate criterion can assess whether the agent performed enough verification, if the holistic rubric requires it.

Explain how to judge it

Use `elaboration` for verification instructions, evidence to inspect, examples, and guidance on partial fulfillment. Explain acceptable alternatives so the grader can credit valid approaches that differ from the original attempt.

For every `dodged_bullet` criterion, describe a similar behavior that is acceptable and should pass. For a criterion about false test claims, explicitly saying that tests were not run is an acceptable alternative.

Categories

The category identifies the kind of requirement. Severity determines its scoring weight.

Category	Meaning
<code>primary_intent</code>	A core task requirement.
<code>extra_credit</code>	Optional behavior that earns additional credit. Its absence is not penalized; omit <code>severity</code> .
<code>dodged_bullet</code>	A mistake the response must avoid. The criterion passes when that mistake does not occur.

Severity

Use the YAML value in the file. The display name is the shorter label used in grading results.

YAML value	Display name	Use
<code>crux</code>	Crux	The central failure, derived from a holistic penalty aimed at the overall score. Most tasks have one; never use more than two.

YAML value	Display name	Use
<code>certain_dealbreaker</code>	Critical	A decisive failure. Fabricated or inaccurate claims about the agent’s own actions default to this level.
<code>possible_dealbreaker</code>	Major	A serious shortfall whose importance depends on the rest of the attempt. An acknowledged but unresolved verification gap defaults to this level.
<code>unlikely_dealbreaker</code>	Minor	A real but rarely decisive concern, such as style, concision, or over-engineering.

How grading works explains severity weights and how criterion verdicts contribute to the score.

Converting holistic penalties

An overall-score penalty becomes one Crux criterion. A penalty aimed only at a named dimension becomes Critical. Do not promote a criterion to Crux merely because low-scoring runs failed it.

If a penalty applies only when two conditions occur together, keep those conditions in one criterion. For example, a penalty for both leaving an authorization flaw unresolved and claiming it is fixed must retain both conditions; splitting them would change when the penalty applies.

Preserve the acceptable behavior described alongside the holistic penalty. Criteria must not describe score arithmetic or contain numeric penalty language.

Rubric examples

These examples show how task-specific guidance becomes atomic criteria. Each package includes a holistic rubric, its atomic conversion, and graded reference runs. Start with the holistic rubric, compare its requirements with the atomic rubric, then look at how both methods graded the reference runs. The score comparisons are in [rubric-regrades/index.json](#).

Use these examples to learn the approach, then develop your own task rather than copying an example. See Write the holistic rubric and Atomic rubric criteria for the current guidance.

Filenames in the examples

These packages use older filenames:

File in the examples	Current equivalent
tests/grader-guidance-consolidated.md	tests/holistic-rubric.md
tests/rubrics.yaml	tests/atomic-rubric.yaml

ZenBill: impersonation permissions

Task: Let managers impersonate non-manager teammates in a payment application, while preventing an impersonated account from restoring access across tenants through its separate platform-admin permissions.

What to look for: The holistic penalty applies only when multiple conditions occur together. The atomic rubric preserves those conditions in a single Crux criterion so the conversion does not change when the penalty applies.

Download example package

Palolo: ACH effective dates

Task: Control when paychecks are routed using an ACH effective-date rule: one banking day early, calculated in Pacific time.

What to look for: The rubric makes the date and time-zone expectations precise. Its central penalty applies when the agent both uses an incorrect raw UTC comparison and confidently claims the work is complete. The atomic criterion retains both conditions.

[Download example package](#)

Zeta: tax-document retries

Task: Add a nightly job to file ready tax documents and retry failures. An existing adapter hides connection failures, which can make retries ineffective or cause duplicate filings.

What to look for: The holistic rubric explains exactly when its penalty applies, including a later retraction that does not undo the mistake, and which behavior avoids the penalty. The atomic rubric preserves that distinction and separately offers extra credit for identifying a notification problem.

[Download example package](#)

Casa: court-report review

Task: Review a court-report feature and decide whether it is ready to merge. Its tests and main flows work, but retry, notification, and deletion paths still contain defects.

What to look for: The rubric distinguishes real defects from plausible but incorrect findings. It also shows that a rubric need not contain a Crux criterion: this holistic rubric has no penalty aimed at the overall score.

[Download example package](#)

Stocks in the Future: support roles

Task: Add a support-staff user type to a system with both a user-type field and a separate admin flag. Some combinations create unsafe permissions that the agent should recognize.

What to look for: The rubric gives credit to two acceptable approaches: asking for clarification before proceeding, or addressing the unsafe admin flag while implementing the new role. Its penalties apply to individual grading dimensions and become Critical criteria in the atomic rubric.

[Download example package](#)

Troubleshooting

Find the error message or symptom below. Toolkit scripts run from the toolkit root inside the Authoring container; container startup commands run on your host machine.

Diagnose it yourself first

Most problems you hit here are ordinary software problems, and you are expected to work them yourself before asking for help. Read the error in full: it usually names the failing command, file, or step, and the fix follows from there. Then check this page for the symptom.

The agents in the toolkit are good at this, so use them. Start Claude Code or Codex in the container where the problem occurred, paste the complete error along with the command you ran, and ask it to explain the failure and what to check next. Let it inspect the relevant files and configuration, try its suggestions one at a time, and confirm whether each one changed the result. Most setup, container, script, and dependency failures are resolved in a few minutes this way.

Keep applying your own judgment while you do it. An agent will confidently propose a fix that does not apply to your situation, so check that its explanation matches what you actually observe before acting on it, and avoid changes you do not understand, particularly ones that alter the task’s starting environment or its saved runs.

Most of what goes wrong is specific to your own machine: your Docker installation and its resource limits, your operating system, your disk space, your network, the editor that saved a file. The project team cannot see any of that and cannot debug a setup that is not in front of them, so asking generally returns the same suggestions you would get faster from the agent. You and the agent in that container are the only ones who can actually inspect the failing machine, which makes you the right people to fix it.

Escalate a clear toolkit defect: something that would fail the same way for anyone, such as a script that is broken for everyone, a file missing from the package, or a failure you can describe precisely that returns after a fix that should have worked. Report it with what you found and what you ruled out. Anything rooted in your own environment stays with you and the agent.

Setup and containers

Problem	What to check
Docker is not running	Start Docker Desktop before starting a container.

Problem	What to check
A container build fails	Read the build error for the failing step. Confirm Docker has at least 4 GB of memory allocated and the host has at least 16 GB of RAM. See the specific build errors below.
Docker runs out of disk space	Check Docker's disk usage with <code>docker system df</code> on your host. If unused resources are taking up space, <code>docker system prune</code> can remove stopped containers, unused networks, dangling images, and build cache. Review its confirmation prompt before proceeding.
Windows setup is slow or mounts fail	Use WSL 2. Copy the toolkit ZIP into the native WSL filesystem and extract it there, outside <code>/mnt/c</code> . For a trial that fails on a mount, see Windows: a trial fails on a mount.
Intel Mac containers will not start	Intel Macs have known limitations. After checking Docker, ask in project Slack for help.
Toolkit ZIP is corrupt	Delete it and download again. Report repeated corruption.
Extracted toolkit is missing <code>explore/repos</code> or has broken links	Re-extract with <code>unzip</code> or 7-Zip. Some built-in extractors drop symbolic links.
A container exited	On your host, run <code>npx @devcontainers/cli up</code> from <code>explore/</code> for Explore or from the toolkit root for Authoring.
<code>harbor-run</code> is not found	Run <code>scripts/harbor-run</code> from the toolkit root inside Authoring. Snapshot commands are available in Explore.
Scripts fail with <code>permission denied</code> on WSL	From the toolkit root, run <code>chmod +x scripts/*</code> .
Rails image fails during Bootsnap setup	For a Bootsnap-specific error, the project's documented workaround is to add <code>DISABLE_BOOTSNAP=1</code> to the toolkit's <code>.env</code> , then recreate the container.
Docker reports an unknown instruction or fails on a heredoc	Update Docker Desktop so the build uses a current BuildKit.
<code>npm</code> or esbuild was installed for another platform	Run toolkit scripts inside Authoring. If the error occurred there, run <code>npm install</code> from the toolkit root and retry.

Problem	What to check
Build stops at <code>could not read Username for 'https://github.com'</code>	The project documents an intermittent GitHub HTTP/2 issue. For that issue, run <code>git config --global http.version HTTP/1.1</code> in the environment running the failing Git command and retry. This changes Git's global HTTP setting in that environment. If the error persists, report it rather than assuming it has the same cause.

Windows: a trial fails on a mount

On Windows with WSL 2, a trial can fail with `mounts denied`, a message that the path is not shared from the host and is not known to Docker, or an empty workspace and no reward file.

Authoring does not run its own Docker. It uses the host's Docker through a shared socket, so every path a trial mounts is resolved by Docker on the host, not inside the container. Authoring is built for this: the toolkit is mounted a second time at its host path, and `scripts/harbor-run` switches to that path before starting a trial. A mount failure means the path Docker was given is not one it can resolve.

Check this before spending another trial. In the Authoring terminal:

bash Copy

```
echo "HOST_WORKSPACE=$HOST_WORKSPACE"
docker run --rm -v "$HOST_WORKSPACE:/probe:ro" alpine:3.20 \
  sh -c 'test -f /probe/scripts/harbor-run && echo "mounts resolve" || echo "mount is empty"'
```

A mount error, or `mount is empty`, confirms the cause. Then check, in order:

- `HOST_WORKSPACE` is set. An empty value means the container was not created from the toolkit's own devcontainer configuration. Recreate it from the toolkit root.
- The toolkit is in the native WSL filesystem, not under `/mnt/c`. Re-extract it there and recreate the container.
- Docker Desktop's WSL integration is enabled for your distribution, and you start containers from that distribution rather than from PowerShell.

If the probe still fails after those checks, work it with the agent in the Authoring terminal as described in Diagnose it yourself first. Give it the probe command and its full output, the value of `HOST_WORKSPACE`, your distribution name, and where the toolkit is extracted, and have it inspect how the container is mounted. Installing Harbor on the host and running trials outside Authoring is not a supported workaround: the toolkit pins the Harbor and Node versions your reference runs are produced with, and a hand-built host environment drifts from them.

Authentication

Problem	What to check
<code>401</code> , <code>invalid_api_key</code> , or “task no longer active”	Confirm you have an active project task. The bundled key is tied to active work on the platform. Download a fresh toolkit if the key remains invalid.
<code>apiKeySource: none</code> or Claude cannot connect	Check that the toolkit’s <code>.env</code> contains <code>ANTHROPIC_API_KEY</code> , <code>ANTHROPIC_BASE_URL</code> , and <code>USER_ID</code> . Restart the container, then check that the variables are populated using the command below.
Claude reports that <code>ANTHROPIC_API_KEY</code> takes precedence over an Anthropic Console key	Expected. Claude is using the API-key authentication supplied by the toolkit.
Codex reports failed WebSocket attempts, then connects over HTTPS	The HTTPS fallback is expected. No change is needed if the connection succeeds.
Codex keeps returning <code>401</code> with an active task	Check for accidental spaces in <code>.env</code> and confirm the base URL contains <code>app-11mproxy</code> . If the file was saved on Windows, its CRLF line endings leave a carriage return on the end of the key, which the model service rejects. See Windows line endings in <code>.env</code> .

Run this in the affected container’s terminal to check whether the variables are populated without printing their values:

bash Copy

```
for name in ANTHROPIC_API_KEY ANTHROPIC_BASE_URL USER_ID; do
  if printenv "$name" | grep -q '^[[:space:]]'; then
    printf '%s is set\n' "$name"
  else
    printf '%s is missing or empty\n' "$name"
  fi
done
```

This checks presence, not whether the credentials are valid.

Windows line endings in `.env`

A `.env` saved by a Windows editor ends every line with CRLF. The shell keeps the carriage return in the value, so the key is sent with an extra character on the end and the request fails with a `401`. Check the file in the affected container:

bashCopy

```
grep -q $'\r' /workspace/.env && echo 'CRLF line endings found' || echo 'line endings are fine'
```

Convert it in place, then restart the container so every process picks up the corrected values:

bashCopy

```
sed -i 's/\r$//' /workspace/.env
```

Explore copies the toolkit root's `.env` into the container each time it starts, so convert the file at the toolkit root as well. Otherwise the next start brings the carriage returns back.

Trials and grading

Problem	What to check
Trial fails on a mount, or produces no reward file, on Windows	See Windows: a trial fails on a mount.
Trial returns a <code>529</code> overload error	Retry after the model service recovers. An overload error does not establish a problem with the task.
Trial times out	Check the trial logs to see whether the agent or verifier timed out. If the agent was interrupted, run a new trial. If the agent finished and only grading failed, use the grading recovery steps below.
Grading errors or produces no score	See Recover a failed grade.
Every run scores at least <code>0.9</code>	This is a warning that the task may not be demonstrating a meaningful failure. Inspect the agent's work to check whether the failure occurred. If it did, investigate whether the grader missed it.

Problem	What to check
Every run scores near 0.0	Inspect the agent’s work and the grader’s reasoning. Check whether the rubric’s factual claims are correct. Use the sections below to investigate.
reward-correctness.txt is N/A	Expected. Correctness is evaluated under Narrow Correctness and Broader Correctness.
Inputs appear truncated	Check whether the input was cut off by the tooling or the agent exhausted its context while working. Retry a tooling failure and report it if it persists. An agent running out of context can count as a meaningful failure if it meets the usual failure criteria. Input cut off by the tooling is an environment problem.
Agent refuses a benign security task	State the defensive purpose clearly. If the refusal persists during calibration, use the provided fallback model when available. Record the refused prompt, response, and fallback as required on the active task, and report repeated refusals.

Recover a failed grade

Every requested grading sample must produce a valid score. If grading fails after the agent finishes, check that the attempt’s transcript and any changed files were captured. You can regrade the saved attempt without asking the agent to do the task again. Use the atomic regrading instructions if the failure occurred during atomic grading.

If the agent was interrupted, or its work was not captured, run a new trial. Regrading cannot recover a missing attempt or missing file changes.

For an error saying the installed Claude Code does not support the grader model, check Toolkit updates. The current toolkit checks for outdated Claude Code in task images before local trials and rebuilds affected images.

The failure does not recur

Inspect the work in each reference run to determine whether the meaningful failure occurred. If it did but the grader missed it, follow the grading guidance below.

If the failure did not occur, check whether the prompt, recorded conversation, or starting files give away the answer, such as comments or README text that explain the fix. Remove unintended hints while keeping the task realistic. Then return to Develop a task to refine the request or investigate another failure. The hardness ladder offers ideas for making realistic tasks more challenging.

A complete submission needs reference-run evidence of the meaningful failure. After editing, follow What to rerun after edits and inspect the new results.

The grade does not match the work

Compare the grader’s explanation with the agent’s trajectory, changed files, and verified repository facts. If rubric guidance is missing or inaccurate, correct it and update the affected grades.

If the rubric accurately describes the requirements and the grader still misjudges the work, report the grading issue through the process shown on the active task. Include the observed behavior and the part of the grade that is incorrect.

Reference-run validation

A complete submission requires at least four reference runs in which the agent finished its attempt, including evidence of the meaningful failure. Replace attempts interrupted by API errors, a non-zero agent exit, or an agent timeout. An agent finishing with an incorrect solution still provides a valid attempt to review.

The current packaging validator handles run counts as follows:

Saved runs	Validator behavior
None	Blocks packaging.
One to three	Warns about the count. This does not meet the complete-submission requirement.
Four or more	Blocks packaging if fewer than four attempts finished without an agent/API execution error.

A verifier-side timeout does not disqualify a completed agent attempt from this count. Check that the required grades are available and reviewed; if grading failed, recover the grade. Passing the run-count check alone does not establish that grading is complete.

Workspace and rebuilds

Problem	What to check
<code>harbor-run</code> reports a workspace mismatch	Compare the local workspace with the intended starting state. If the differences belong in the task’s starting environment, update the patch. If they came from the agent’s attempted solution or unrelated edits, restore the intended starting state before running trials.

Problem	What to check
Intended starting-workspace changes do not appear in trials	Regenerate <code>workspace.patch</code> , then rerun with <code>--force-build</code> . See Prepare the task environment.
<code>workspace/: No such file or directory</code>	Run <code>bash scripts/build-workspace.sh <slug></code> .
Unsure whether a holistic-rubric edit needs <code>--force-build</code>	It does not. Regrade the saved runs; the grader reads the rubric fresh. See What to rerun after edits.
Trial fails with <code>UNKNOWN_CERTIFICATE_VERIFICATION_ERROR</code>	Check whether <code>allow_internet</code> in <code>task.toml</code> was changed to <code>false</code> . If so, restore the default <code>true</code> ; the agent needs network access to reach the model. Report the error if it persists with the default setting.

Atomic rubric and regrading

Problem	What to check
<code>stage-atomic-rubric.ts</code> finds two atomic rubric files	Keep the filename the task was created with: <code>atomic-rubric.yaml</code> for current tasks or <code>rubrics.yaml</code> for older ones. Compare the files and preserve any needed edits before removing an accidental duplicate. See Keep existing filenames.
Atomic and holistic scores differ sharply	Compare criterion verdicts with the observed work, then follow the grading comparison guidance.
Regrade job folders collide	Give each parallel job a distinct <code>HARBOR_REGRADE_OUT=harbor-jobs/<run></code> . With 4 GB of Docker memory, run no more than one or two at once.

Packaging

Read the errors and warnings from `submit-task.ts` . Errors block archive creation. Warnings allow packaging to continue, but reviewers can see them; investigate each one before submitting.

Problem	What to check
Required task files are missing	Compare the task folder with Task files and complete the relevant authoring step.
The example source path does not exist on your machine	Set <code>trial</code> to your actual completed atomic trial folder containing <code>result.json</code> and <code>verifier/</code> . Look for <code>reference_run_dir</code> in its <code>config.json</code> to find the original reference run's folder name for the destination. Existing valid grades do not need to be moved or regenerated. See Store the atomic grades.
A reviewer returns a complete submission for missing atomic grades	Grade each reference run under the atomic rubric; each grade is stored in <code>rubric-regrades/<run>/</code> as it finishes. Then rebuild the archive. If the task is already out for review, wait for feedback and add them in that revision.
The declared agent does not match the reference runs	Check <code>[agent].harness</code> in <code>task.toml</code> against the agent application that produced the runs. All reference runs must use the task's configured agent.
<code>submit-task.ts</code> reports placeholders	Search <code>instruction.md</code> and the holistic rubric for <code>Replace this</code> , then replace every scaffold marker.
Detector warnings appear	Read the named detector's report. Run it if the report is missing, or rerun it once after the last edit if its inputs changed. Fix real problems; report incorrect findings as false positives on Surge's Test page. See Review the findings.
A session contains the snapshot command	Remove the snapshot-command entries from <code>environment/session.jsonl</code> and check that the history does not reveal the answer. Because the recorded conversation changed, rerun the affected trials and detectors.
A run is stale after an edit	Follow What to rerun after edits.
Packaging reports too few completed reference runs	Follow Reference-run validation.
Temporary rubric grading files appear in the archive	Restore the staged files, then package again.

Still blocked?

The project FAQ has additional guidance. Ask general questions in [#ext-surge-raccoon-questions](#) .

Ask after you have investigated the failure yourself, and include what you found: the exact error, the command that produced it, where it ran, and what you have already tried and ruled out. That is also what makes a problem reproducible for whoever answers.

Repository catalog

Choose a codebase with a familiar language, business domain, and area of engineering. For a single table of every repository with its language and a one-line description — including each repository inside the polyglot toolkits — see the full repository appendix. This catalog describes the toolkits across the project; use the downloads available on your task page. See Choose a repository for selection guidance. Some toolkits are available only for tasks already created on them; they are listed under Toolkits available only for existing tasks.

Compare toolkits

Toolkit	Application or domain	Main technologies	Source
ZenBill (ZenBill-006)	Business payments and invoicing	Ruby, Rails, React, PostgreSQL	Private
Palolo (Palolo-031)	Employee financial wellness and payroll	TypeScript, Express, Prisma, React	Private
zeta-platform	Legacy banking platform	Ruby, Rails, React, PostgreSQL, Redis	Private
zeta-polyglot	Banking services and related applications	Ruby, Node.js, React, Python; varies by repository	Private
Breezy (breezy-complete)	AI phone receptionist for home-service businesses	Ruby, Rails, Next.js, PostgreSQL, Redis	Private
Potion (potion-polyglot)	Personalized video for sales and marketing	JavaScript, Nuxt/Vue, Express, MongoDB, Python	Private
Speedwell (speedwell-polyglot)	Member concierge for families and households	TypeScript, Remix, Elixir/Phoenix, Node.js, PostgreSQL	Private
casa	Court-appointed volunteer advocacy	Ruby, Rails, PostgreSQL, RSpec	Open source

Toolkit	Application or domain	Main technologies	Source
freeitsm	IT service desk, CMDB, and asset management	PHP, Apache, MySQL	Open source
frepple	Supply chain planning: forecasting, production, and inventory	C++, Python (Django), Vue, PostgreSQL	Open source

Framework and runtime versions are pinned in each toolkit. Check its README and dependency files for the exact versions.

Repository details

ZenBill ([ZenBill-006](#))

A business payment and invoicing application with a Rails backend and React frontend. Work includes ACH and card payments, invoices, permissions, and accounting synchronization. Authorization is tied to a user's membership in an organization. Integrations include Dwolla, Plaid, Finix, QuickBooks Online, and Stripe.

Read the ZenBill codebase tour.

Palolo ([Palolo-031](#))

A TypeScript monorepo for earned wage access, loans, savings, and payroll integrations. It includes separate interfaces for employees and employers, an Express API, background jobs, and multi-factor authentication. The codebase connects to banking and payroll providers such as Atomic, Finch, and Argyle.

Read the Palolo codebase tour.

zeta-platform

A large legacy Rails banking application covering cards, ACH risk, virtual card issuance, and member notifications. It includes React code within the repository and uses PostgreSQL and Redis. This toolkit may suit authors familiar with older Rails applications and financial systems. The API and its tests run without the frontend, so passing those tests does not verify frontend behavior.

zeta-polyglot

A toolkit containing multiple repositories from the wider Zeta ecosystem, including Rails services, Node.js and React applications, Python projects, and repositories intended for inspection rather than running an application. Languages and setup requirements vary, so choose a repository within the toolkit that matches your experience — the full repository appendix lists all thirty-eight.

Breezy ([breezy-complete](#))

An AI phone-receptionist platform for home-service businesses, with a Rails API and Next.js frontend. The application covers calls, SMS, scheduling, campaigns, billing, and website content. It includes background jobs and integrations with communication and payment providers.

Potion ([potion-polyglot](#))

A toolkit containing multiple repositories for personalized video and avatar generation. The main application uses Nuxt/Vue, Express, and MongoDB. Other repositories cover Python inference services, video-processing workers, screen recordings, landing pages, billing, and cloud infrastructure. The full repository appendix lists all fifty-two.

Speedwell ([speedwell-polyglot](#))

A toolkit containing eighteen repositories for a member-concierge product, where specialists manage contacts, important dates, projects, and documents on behalf of member families. The main application is a Remix and Prisma front end in TypeScript, backed by an Elixir/Phoenix service that handles messaging, calendar accounts, and date recommendations; `run-app strongsuit-app` starts both. The rest of the estate is an earlier Express and MongoDB API with its React client, two small agenda clients, three browser extensions, a Go job-queue wrapper, Terraform for Google Cloud, and several short Node and Python integrations.

Sign-in goes through a hosted identity provider that cannot be reached offline, so the toolkit ships a local dev-login route; `run-app` prints the URL to use.

casa

A Rails case-management application for court-appointed volunteer advocates. Work includes case records, volunteer coordination, and court reports. The repository has an extensive RSpec suite, including tests that exercise the application through the browser.

frepple

Open-source supply chain planning: demand forecasting, production planning, material and capacity constraints, distribution, and inventory. A C++ planning engine does the work, and a Django web application with a Vue frontend is how planners drive it. Work spans all three — the engine's model of buffers, calendars, operations and resources; the Django app's screens, permissions and data import;

and the frontend. It is the only toolkit that combines C++, Python and Vue in a single codebase, and the only supply chain domain in the project.

Two test suites. The engine's is 83 named scenarios under `test/`, each paired with an expected-output file — `constraints_leadtime_*`, `constraints_material_*`, `constraints_resource_*`, `buffer_max`, `demand_policy` — and it runs in seconds without a database. The Django suite takes a few minutes and needs PostgreSQL. One browser test (`freppledb.forecast.tests.test_frontend`) fails in the container because it needs Chrome; that is expected and is not a problem with your setup.

The planning engine is rebuilt with `cmake --build build --parallel` after you edit `src/`. Forecasting (`src/forecast`) is statistical rather than rule-based, so it is a poor choice of subject for a task; the model and the Django application are the better ground.

freeitsm

A self-hosted IT service management application: tickets with service levels, a configuration management database, change management, an asset register, contracts, checklists, a self-service portal, tasks, a calendar, and a knowledge base. It is plain PHP served by Apache over MySQL, with no framework and no Composer, so behavior is read from the code itself rather than inferred from framework conventions. It is the only PHP toolkit in the project, and it may suit authors who prefer service-management and operations work to consumer or financial applications.

Its checks are 27 standalone PHP scripts under `tests/` rather than a test framework, run one at a time and as the `www-data` user. Two of them, `calendar-sync-tasks.php` and `task-recurrence-spawn.php`, exit with an error and print nothing; that is a pre-existing defect in those two scripts and not a problem with your environment. Because they do not run, avoid tasks whose correctness depends on calendar synchronization or task recurrence, where no check would confirm your agent's work.

Toolkits with multiple repositories

Zeta Polyglot and Potion Polyglot bundle several repositories. Each task still targets exactly one repository; the trial agent cannot inspect the others. The full repository appendix lists every member repository with its language and purpose.

Use Set up the toolkit to initialize a repository and Prepare the task environment to configure the task. Trial environment explains repository paths and access during trials.

Zeta reference corpus

The Zeta toolkits include project discussions, support conversations, email, and issue history that can help you understand application behavior and business context. This material is available at `/data/zeta-corpus/` in Explore and during trials.

Refer to the supplied paths directly. The toolkit makes the corpus available when the task is rebuilt, so you do not need to copy it into the repository or submission. See Trial environment for files available to the agent.

Toolkits available only for existing tasks

These toolkits are deprecated. Do not start a new task on one. They remain in the project so that tasks already created on them can be finished, revised, and reviewed; if you have such a task in progress or returned for edits, keep working in it as normal.

Toolkit	Application or domain	Main technologies
human-essentials	Inventory and distribution for essentials banks	Ruby, Rails, PostgreSQL, RSpec
awbw	Art programs for trauma recovery	Ruby, Rails, MySQL, Vite, RSpec
stocks-in-the-future	Financial literacy and simulated portfolios	Ruby, Rails, PostgreSQL, Redis, Minitest
community-foundation	Fund planning and allocation	Ruby, Rails, SQLite, Minitest
endsideout	Sports and education programs	Ruby, Rails, SQLite, Minitest
flaredown	Chronic-illness symptom tracking	Ruby, Rails API, Ember, MongoDB, PostgreSQL

human-essentials

An inventory and distribution application for diaper banks and other essentials banks. Its Rails code handles donations, purchases, requests, partner organizations, and inventory movements, with data separated by organization. Inventory is tracked through recorded events.

awbw

A Rails application supporting art programs for trauma recovery. It uses MySQL and Vite and includes payment and geocoding integrations. It may suit authors familiar with program-management applications and Rails frontend work.

stocks-in-the-future

A Rails application for classroom financial literacy and simulated investment portfolios. It uses PostgreSQL, Redis, and Minitest, with work spanning portfolio behavior and user roles.

community-foundation

A smaller Rails application for fund planning and allocation. It uses SQLite and Minitest, with a test-coverage requirement built into its checks.

endsideout

A smaller Rails application for sports and education programs. It uses SQLite and Minitest and includes browser-based system and accessibility tests.

flaredown

A symptom-tracking application with a Rails API and Ember frontend. It combines MongoDB and PostgreSQL and includes background processing. It may suit authors familiar with API/client applications and health-related data tracking. The verifier does not run browser or acceptance tests.

Full repository appendix

Every repository you can choose from, with its main language and a one-line description. The three polyglot toolkits bundle many repositories each; this page is the only place their contents are listed individually.

Languages are each repository's own primary language at the commit your toolkit checks out. Where a repository is a library, a set of notebooks, or configuration with nothing to boot, the description says so — you can still read it, but you cannot run an application in it.

For a toolkit-level comparison and longer descriptions, see the repository catalog. For guidance on picking one, see Choose a repository.

Languages at a glance

Language	Where to find it
Ruby	ZenBill, Zeta Platform, Breezy, CASA, and the Rails services in zeta-polyglot
TypeScript	Palolo, ZenBill, Breezy, the main application in speedwell-polyglot , front ends in zeta-polyglot , analytics in potion-polyglot
JavaScript	Most of potion-polyglot ; the React front ends in zeta-polyglot ; the older clients and integrations in speedwell-polyglot
Vue and Nuxt	The main applications in potion-polyglot ; frePPLe's frontend
Python	frePPLe's Django web application; the AI agents in zeta-polyglot ; the AI, voice, and video services in potion-polyglot ; small scripts in speedwell-polyglot
SQL and dbt	The analytics warehouse in zeta-polyglot
C++	frePPLe's planning engine (the only C++ in the project)
PHP	FreeITSM (the only substantial PHP in the project); one WordPress site in potion-polyglot

Language	Where to find it
Terraform	The infrastructure repositories in potion-polyglot and speedwell-polyglot
React Native	The mobile client in zeta-polyglot (tests only; the app is not booted)
Elixir	The Phoenix service in speedwell-polyglot (the only Elixir in the project)
Go	One job-queue service in speedwell-polyglot (tests only; the service is not booted)

Single-repository toolkits

Each of these toolkits contains one repository, checked out at [repo/](#).

Repository	Toolkit	Language	Description
ZenBill-006	ZenBill	Ruby (Rails) and TypeScript (React)	Business payments and invoicing: ACH and card payments, invoices, organisation permissions, and accounting synchronisation.
Palolo-031	Palolo	TypeScript (Express, Prisma, React)	Employee financial wellness: earned wage access, loans, savings, and payroll integrations, with separate employee and employer interfaces.
zeta-platform	Zeta Platform	Ruby (Rails, legacy)	Large legacy banking application: cards, ACH risk, virtual card issuance, and member notifications. React code lives in the repository, but the API and its tests run without the frontend.
breezy-complete	Breezy	Ruby (Rails API) and TypeScript (Next.js)	AI phone receptionist for home-service businesses: calls, SMS, scheduling, campaigns, billing, and website content.
casa	CASA	Ruby (Rails)	Case management for court-appointed volunteer advocates: case records, volunteer coordination, and court reports. Extensive RSpec suite, including browser tests.

Repository	Toolkit	Language	Description
freeitsm	FreelTSM	PHP (no framework)	Self-hosted IT service desk: tickets with service levels, a configuration management database, change management, an asset register, contracts, and a self-service portal. Plain PHP over Apache and MySQL, with no framework and no Composer.
frepple	frePPLe	C++ (engine), Python (Django), Vue	Open-source supply chain planning: forecasting, production planning, material and capacity constraints, distribution, and inventory. The C++ engine plans; the Django application and Vue frontend are the interface. Two suites — 83 engine scenarios with expected-output files, and a Django suite needing PostgreSQL.

Repositories in [zeta-polyglot](#)

Thirty-eight repositories from a consumer banking ecosystem: Rails services, React and Next.js front ends, Python AI-agent libraries, the analytics warehouse, and several documentation-only repositories.

Ruby and Rails services

Repository	Toolkit	Language	Description
zeta-wasabi-platform	zeta-polyglot	Ruby (Rails)	The main consumer banking application: personal financial management, multi-database, with an IVR phone surface. The largest Rails member.
zeta-heimdall	zeta-polyglot	Ruby (Rails API)	Authentication and identity service for the banking platform: sign-in, tokens, and account access. Postgres.
zeta-px-api	zeta-polyglot	Ruby (Rails API)	Account-linking API built to an external aggregator specification, so third-party services can connect to joint card accounts.
zeta-plastic	zeta-polyglot	Ruby (Rails)	Card service: cards and card data, account ownership, and device registrations. Dockerised, with Rubocop and GitHub Actions.

Repository	Toolkit	Language	Description
zeta-hook	zeta-polyglot	Ruby (Rails)	Inbound webhook receiver for third-party events. Postgres and Redis, with Karafka consumers.
zeta-bureau	zeta-polyglot	Ruby (Rails)	Vendor identification: matches and classifies transaction vendors, learning from earlier matches. Small, with an admin surface.
zeta-hive	zeta-polyglot	Ruby (Rails)	Subscriber and mailing-list management: lists and memberships, unsubscribes, email logs, reviews, and content engagement.
zeta-hiddentemple	zeta-polyglot	Ruby (Rails)	Core money-movement and ledger service: accounts, ACH, checks, card transactions and events, holds, and counterparties.
zeta-switchboard	zeta-polyglot	Ruby (Rack)	Request router in front of the public help domain: forwards each path to the right upstream service, and generates the nginx config.
zeta-mule-deprecated	zeta-polyglot	Ruby (Rails)	Retired analytics and attribution service for the joint-card product, kept for reference.
zeta-jc-loadtester	zeta-polyglot	Ruby (CLI)	Load-testing harness that scripts repeatable Apache Benchmark runs as version-controlled Ruby test files. Library and CLI, tests only.

JavaScript and TypeScript front ends

Repository	Toolkit	Language	Description
zeta-wasabi-next	zeta-polyglot	TypeScript (Next.js)	Newer customer web front end on Next.js, with Apollo GraphQL and Sentry.
zeta-wasabi-web	zeta-polyglot	JavaScript (React, CRA)	Customer-facing web client for the banking platform, built with Create React App.
zeta-bureau-ui	zeta-polyglot	JavaScript (React, CRA)	Front end for the back-office administration tool. Apollo GraphQL client, Chakra UI, easy-peasy state.

Repository	Toolkit	Language	Description
zeta-jarvis	zeta-polyglot	JavaScript (React, CRA)	Internal customer-support console over the main banking platform: accounts, cards, conversations, complaints, change requests, and internal adjustments.
zeta-wasabi-mobile-bare	zeta-polyglot	TypeScript / JavaScript (React Native)	The iOS and Android mobile banking client. Tests run, but the app itself is not booted in the container.
zeta-content-hub	zeta-polyglot	JavaScript (Gatsby)	Static content site generated from a headless CMS, with Apollo and Chakra UI.
zeta-blog	zeta-polyglot	JavaScript (Gatsby)	Marketing blog generated from a headless CMS, derived from a Gatsby starter.
zeta-web-code-challenge	zeta-polyglot	TypeScript (Next.js)	Small self-contained hiring exercise: a peer-to-peer payments UI over a mock JSON API. Good for a compact, low-context task.

Python AI agents and services

Repository	Toolkit	Language	Description
zeta-cx-chatbot	zeta-polyglot	Python (Flask)	LLM-backed first-line support assistant for the customer-experience team, served over an authenticated HTTP endpoint with a background worker.
zeta-compiler-agent	zeta-polyglot	Python / SQL (library)	Planning agent that decomposes a question into tasks, runs them in parallel, and assembles the answer.
zeta-agent-tools	zeta-polyglot	Python / SQL (library)	Shared tool library the AI agents call, including the governed data-access queries behind them.
zeta-agent-tool-kb-search	zeta-polyglot	Python (library)	Retrieval tool that searches the support knowledge base for an agent, following a self-RAG pattern over LangGraph.

Repository	Toolkit	Language	Description
zeta-agent-tool-bill-goal-ledgers	zeta-polyglot	Python / SQL (library)	Agent tool that retrieves a customer's bill and savings-goal ledger information.
zeta-chatbot-embeddings	zeta-polyglot	Python (scripts)	Builds the embeddings and vector store that the support chatbot retrieves against.

Data, analytics, and machine learning

Repository	Toolkit	Language	Description
zeta-dbt	zeta-polyglot	SQL / Python (dbt)	The analytics warehouse: dbt models, macros, seeds, and custom lint and scoring rules. SQL modelling work, not an application.
zeta-predictors	zeta-polyglot	Jupyter / Python	Data-science package of predictive models, with exploratory notebooks alongside the packaged code.
zeta-transaction-anomalies-ml	zeta-polyglot	Jupyter / SQL	Exploratory notebooks on transaction anomaly detection, with the SQL that feeds them. Analysis rather than a service.
zeta-snowflake-admin	zeta-polyglot	SQL	Warehouse administration expressed as SQL: roles, grants, and the setup for each connected tool.
zeta-data-metadata	zeta-polyglot	Documentation only	Notes on the data warehouse itself — replication behaviour, type handling, and glossary terms that do not belong in the dbt project.

Documentation, templates, and infrastructure

Repository	Toolkit	Language	Description
zeta-docs	zeta-polyglot	Documentation only	Cross-service engineering documentation: how each system works and how they connect. The best starting point for understanding the estate.

Repository	Toolkit	Language	Description
zeta-emails	zeta-polyglot	HTML	Transactional email and mailer templates with their shared assets.
zeta-ops	zeta-polyglot	Ruby (scripts)	Small operations scripts, chiefly SSL certificate rotation.
zeta-nginx-buildpack	zeta-polyglot	Ruby (buildpack)	Platform buildpack for running nginx on a hosted dyno. Infrastructure plumbing.
zeta-ada	zeta-polyglot	Documentation only	Stub repository for a data-analyst assistant. Effectively a README; nothing to run.
zeta-rails-challenge	zeta-polyglot	Documentation only	Instructions for a Rails hiring exercise. No application code.
zeta-react-challenge	zeta-polyglot	Documentation only	Instructions for a React Native hiring exercise. No application code.
zeta-react-challenge-internal	zeta-polyglot	Documentation only	Instructions for a React hiring exercise. No application code.

Repositories in [speedwell-polyglot](#)

Eighteen repositories from a member-concierge product: a Remix application and the Phoenix service behind it, two older clients and an Express API, a handful of browser extensions and small integrations, and the infrastructure around them.

Applications and APIs

Repository	Toolkit	Language	Description
strongsuit-app	speedwell-polyglot	TypeScript (Remix)	The main application: members and families, contacts, important dates, projects, documents, and the specialist inbox. The largest member and the estate's centre of gravity.

Repository	Toolkit	Language	Description
strongsuit_phx	speedwell-polyglot	Elixir (Phoenix)	The backend the main application calls for messaging, calendar accounts, and date recommendations. Starts automatically alongside strongsuit-app .
strongsuit-server	speedwell-polyglot	JavaScript (Express)	An earlier Express and MongoDB API. Its tests mock the database and run without one; the server itself needs a MongoDB that is not in the container.
strongsuit-client	speedwell-polyglot	JavaScript (React)	The earlier web client that paired with the Express API.
speedwell-react	speedwell-polyglot	JavaScript (React)	A small Create React App client for building meeting agendas.
speedwell-agenda-builder	speedwell-polyglot	JavaScript (React)	A second Create React App agenda tool. Read-only: no dev server is wired.

Integrations and automations

Repository	Toolkit	Language	Description
strongsuit-zingle	speedwell-polyglot	JavaScript	Creates messaging-platform contacts when a member is added. A few files; nothing to boot.
strongsuit-member-feedback-to-airtable	speedwell-polyglot	JavaScript	Receives a form webhook and writes member feedback to a spreadsheet service. Nothing to boot.
speedwell-node-a2g-sync	speedwell-polyglot	JavaScript	A small Node sync job between two internal systems. Nothing to boot.
hello-sunshine	speedwell-polyglot	JavaScript	Boilerplate for deploying a chat bot. Seven files; a template rather than an application.
strongsuit-arial	speedwell-polyglot	Python	Scripts that read mail and forward it onward. Nothing to boot.

Browser extensions

Repository	Toolkit	Language	Description
strongsuit-chrome-extension	speedwell-polyglot	TypeScript (React)	The current browser extension for specialists, built with React and Jotai.
strongsuit-chrome-extension-OG	speedwell-polyglot	JavaScript	The original capture-tool extension it replaced.
strongsuit-assistant-chrome-plugin	speedwell-polyglot	JavaScript	A small assistant plugin. No README; read the source.

Infrastructure and assets

Repository	Toolkit	Language	Description
jobs	speedwell-polyglot	Go	An API wrapper around a job queue. Its tests run; the service is not booted in the container.
terraform	speedwell-polyglot	Terraform	Google Cloud infrastructure. Validate and format checks rather than an application.
strongsuit-platform	speedwell-polyglot	Python and JavaScript	Version-controlled scripts, configuration, and notes for the wider platform. No single application.
strongsuit-assets	speedwell-polyglot	HTML	Static assets and templates used by systems outside the main application.

Repositories in [potion-polyglot](#)

Fifty-two repositories from a personalised-video and avatar-generation product: the main Nuxt applications and Express API, a large video and voice processing pipeline, the AI inference services behind it, and the infrastructure that runs them.

Applications and APIs

Repository	Toolkit	Language	Description
potion-app	potion-polyglot	Vue (Nuxt 2)	The main personalised-video application: campaigns, recordings, templates, and viewer pages. The largest front-end member and the estate's centre of gravity.
potion-api	potion-polyglot	JavaScript (Express)	The core REST API behind the main application, on Express and MongoDB. Most business logic lives here.
potion-web	potion-polyglot	Vue (Nuxt 3)	Newer Nuxt 3 front end, a rebuild of the main application's web surface.
potion-custom-domain-app	potion-polyglot	Vue (Nuxt 2)	Serves customer pages on their own custom domains, including DNS record checks and certificate provisioning.
potion-analytics	potion-polyglot	TypeScript (Express)	Analytics service that aggregates campaign and viewer engagement metrics. The only TypeScript member.
browser-extensions	potion-polyglot	JavaScript	Chrome extension for recording from the browser, with its own styles and assets.
potion-website	potion-polyglot	SCSS / JavaScript	The public marketing site, built with gulp and webpack. Styling and content rather than application logic.
potion-wp-site	potion-polyglot	PHP (WordPress)	WordPress marketing site. The first-party code is the custom theme; WordPress core itself is vendored.
gcp-application	potion-polyglot	JavaScript	A single-file containerised service with its build config. Very small — read it before choosing it.

Video and voice pipeline

Repository	Toolkit	Language	Description
potion-video-processing	potion-polyglot	JavaScript	The video-processing microservice: transcoding and rendering the generated videos.
lambda-video-processing	potion-polyglot	JavaScript (Lambda)	Lambda entry point into the video-processing pipeline.

Repository	Toolkit	Language	Description
potion-stitch	potion-polyglot	Python	Stitches the recorded and generated segments into one personalised output. FFmpeg, S3, and volume normalisation.
potion-video-background-change	potion-polyglot	Python / JavaScript	Replaces or blurs the background in a recorded video.
microservice-dynamic-screen-recording	potion-polyglot	JavaScript	Service that drives automated screen recordings of a prospect's website for use inside a video.
potion-dynamic-screen-recording-lambda	potion-polyglot	JavaScript (Lambda)	Lambda wrapper around the dynamic screen-recording service.
potion-website-recording-handler	potion-polyglot	JavaScript	Webhook handler that receives completed website-recording callbacks.
microservice-potion-voice	potion-polyglot	JavaScript	Service layer in front of the voice-cloning workers.
potion-voice	potion-polyglot	Python / JavaScript	Text-to-speech and voice-cloning service: multi-speaker baseline training and speech synthesis.
potion-voice-utils	potion-polyglot	Python (library)	Shared helper library for the voice services.
potion-voice-dataset	potion-polyglot	Python	Preparation and bookkeeping for the voice training dataset.
sentence-split-service	potion-polyglot	Python	Transcribes audio with a speech-to-text model and splits the result into sentences, in parallel.
lambda-text-to-speech	potion-polyglot	JavaScript (Lambda)	Thin Lambda wrapper over a third-party text-to-speech API.

Queues, schedulers, and lambdas

Repository	Toolkit	Language	Description
potion-job-producer	potion-polyglot	JavaScript (Lambda)	Lambda that pushes rendering jobs onto the AI queue.
potion-job-consumer	potion-polyglot	JavaScript (Lambda)	Lambda that consumes queued payloads and dispatches them to the AI workers.
potion-watcher	potion-polyglot	JavaScript (Lambda)	Lambda that watches the AI queue depth and triggers the producer. Queue-orchestration logic.
potion-multi-dsr-watcher	potion-polyglot	JavaScript (Lambda)	Watcher for the multi-instance screen-recording workers.
lambda-potion-scheduler	potion-polyglot	JavaScript (Lambda)	Lambdas for the job scheduler.
lambda-potion-transcription-scheduler	potion-polyglot	JavaScript (Lambda)	Schedules video and audio transcription jobs.
lambda-potion-engagement	potion-polyglot	JavaScript (Lambda)	Lambda handling viewer engagement events and subtitles.
lambda-cloudwatch-logs-to-loggly	potion-polyglot	JavaScript (Lambda)	Ships cloud log groups to a third-party log aggregator.
lambda-datadog-forwarder	potion-polyglot	Python (Lambda)	Vendor-supplied log and metric forwarder to the monitoring provider.

AI models and research

Repository	Toolkit	Language	Description
potion-ai	potion-polyglot	Python	The lip-sync inference service, built on the Wav2Lip model. The core of the avatar generation.
potion-ai-gpu	potion-polyglot	Python	GPU deployment of the AI inference service on Kubernetes.

Repository	Toolkit	Language	Description
potion-ai-cpu	potion-polyglot	Python	CPU deployment of the AI inference service as a managed container service.
potion-tryon	potion-polyglot	Python	AI backend for the virtual try-on product.
yeahsure-tryon	potion-polyglot	Python	A second, separate try-on experiment. Little documentation — read the code.
wav2lip-fa	potion-polyglot	Python	In-house fork of the Wav2Lip training code, with multiprocessing added to preprocessing.
MODNet-with-training	potion-polyglot	Python	Fork of the MODNet matting model with the training code adapted for custom datasets.
avds-cleaner	potion-polyglot	Python	Audio/video dataset cleaning routines, derived from an audio-to-video synchronisation network.
avspeech	potion-polyglot	Python	Processing routines for a large public audio-visual speech dataset, used to build higher-resolution training sets.
video-synth-api	potion-polyglot	Jupyter / Python	Modularised deployment of the AI service as separate managed container services, with the notebooks behind it.
urlbox-experiments	potion-polyglot	Python	Standalone spike evaluating a hosted screenshot service as a replacement for the in-house headless-browser capture.

Testing

Repository	Toolkit	Language	Description
potion-qa	potion-polyglot	JavaScript (Selenium)	Browser end-to-end tests driven by Selenium.

Repository	Toolkit	Language	Description
potion-snapshot-testing	potion-polyglot	JavaScript (Playwright)	Visual regression suite that compares rendered screenshots across user tiers.

Infrastructure and deployment

Repository	Toolkit	Language	Description
potion-app-infra	potion-polyglot	Terraform	Terraform for the main application's cloud resources, per environment workspace.
potion-video-processing-devops	potion-polyglot	Terraform	Terraform for the video-processing service: IAM, container registry, and compute.
potion-ai-pretrained-models-infra	potion-polyglot	Terraform / Python	Deploys a function that pulls pre-trained models from an external source into object storage.
potion-bastion	potion-polyglot	Terraform	Bastion-host definitions for each environment.
gcp-infrastructure	potion-polyglot	Terraform	Cloud infrastructure definitions for the second cloud provider.
gcp-cloud-infrastructure	potion-polyglot	Deployment templates	Deployment templates only. Nothing to run; read-only.
potion-devops	potion-polyglot	Dockerfiles / Jenkins	Build and deploy configuration — Dockerfiles and pipelines across four environments. Read-only.
elasticmq-container	potion-polyglot	Dockerfile / config	Container image and queue configuration for a local message-queue stand-in. Configuration only.

Links and resources

Find project tools, supporting guidance, walkthroughs, and repository tours. Most resources require an active Surge or Google session.

Project tools

- Submission Overview: standing, reported hours, queue outlook, and submission history.
 - Task coverage viewer: task counts grouped by repository, integration, and subsystem. Surge currently labels this link "Task Catalog viewer." Download catalog data (JSON)
 - Past responses: view or export earlier submissions. Submit revisions through your task page.
-

Guidance and standards

- Grading Standard: shared grading dimensions and evaluation rules.
 - Engineering workflows: engineering task types grouped by category and priority, including implementation, testing, debugging, and review.
 - Project FAQ: answers to common project questions. **If the link is blocked or doesn't open, right-click and choose Open link in new tab.**
 - Searching for model failures: hardness ladder: guidance for developing more challenging engineering tasks.
-

Walkthroughs and refreshers

- Slack Onboarding Guide
 - Two-minute Raccoon overview (PDF): a short introduction to the project. **If the link is blocked or doesn't open, right-click and choose Open link in new tab.**
 - Task examples discussion (video, about 20 minutes): Nick discusses examples of tasks he likes.
 - Learning Exercise refresher: revisit the learning exercise's tasks and expert answers after completing the exercise on your dashboard.
-

Repository tours

- ZenBill codebase tour (PDF): an introduction to the payment application. **If the link is blocked or doesn't open, right-click and choose Open link in new tab.**
 - Palolo codebase tour (PDF): an introduction to the employee financial-wellness application. **If the link is blocked or doesn't open, right-click and choose Open link in new tab.**
-

Rubric examples

Rubric examples: example holistic and atomic rubrics, with graded reference runs and downloadable packages.

Confidential material

These resources contain confidential project material. Follow the confidentiality requirements when accessing or using them.

Toolkit updates

Start new tasks with the latest announced toolkit. Keep an in-progress task, including one in reviewer feedback, on its existing toolkit unless a release announcement asks you to update. The [4ead3c97eb](#) announcement asks: move every task you have not yet submitted, including one in reviewer feedback, onto [4ead3c97eb](#) with the steps below. If your task was limited to 24 atomic criteria, and your holistic guidance contained more than 24 scoring-relevant rules, regenerate the atomic rubric with [/write-atomic-rubric](#) after you migrate. A task that has already been accepted needs nothing.

Before updating an existing task, read the toolkit update announcement in [#ext-surge-raccoon-announcements](#) and the migration steps below.

If guidance changes during a task

If project guidance changes after you start a task, you may finish using the guidance in effect when you started, unless the change announcement says otherwise. This applies to project instructions as well as toolkit updates.

Check your version

Open [CHANGELOG.md](#) at the toolkit root and compare its top version identifier with the latest release announcement. That identifier is the one releases are announced under, and the only one to compare.

Updating the toolkit does not automatically change an existing task's grader. A task with graded reference runs keeps its pinned grading files unless you deliberately update the grader.

Identifiers you will see

Four values identify a toolkit or a task, and they are not interchangeable. Only the first one tracks releases.

Where	What it is
<code>CHANGELOG.md</code> , the top heading	The release identifier, and the one to compare with a release announcement.
<code>toolkit.json</code> , <code>version</code>	Internal build metadata. Ignore it.
<code>task.toml</code> , <code>[metadata].toolkit_version</code>	The same internal metadata, copied into a task when its folder is created. Leave the line in place; task tooling reads it.
<code>task.toml</code> , <code>[metadata].commit</code>	The repository commit your task starts from.

Both internal stamps record the build your toolkit archive came from rather than the release it belongs to, so they usually differ from the identifier in `CHANGELOG.md` . That is expected, and neither is a way to check your version.

`commit` is taken from the pin that toolkit ships for the repository — `defaultCommit` in `toolkit.json` — at the moment you create the task, and then stays as it is. A later toolkit may ship a different pin, so the `defaultCommit` in a toolkit you move a task into need not match the task’s own `commit` . That difference on its own is fine: your task rebuilds from the state you authored against, which is what makes its reference runs meaningful. Do not edit `commit` to match a newer pin. Changing it changes the starting state and makes the existing reference runs unusable, as What to rerun after edits describes. If building the workspace on a new toolkit cannot find your pinned commit in the supplied repository, report it rather than re-pinning the task.

Keep existing filenames

Keep the grading filenames your task was created with. The toolkit supports the earlier names:

File in an existing task	Name used for new tasks
<code>tests/grader-guidance-consolidated.md</code> or <code>tests/grader-guidance.md</code>	<code>tests/holistic-rubric.md</code>
<code>tests/rubrics.yaml</code>	<code>tests/atomic-rubric.yaml</code>

If you add an atomic rubric for the first time, use `tests/atomic-rubric.yaml` and `tests/grader-context.md` . Do not rename an existing rubric to match the new scaffold.

Move an existing task to a new toolkit

1. **Save the work you need.** Keep the original toolkit until you have verified the copy. Your task lives under `harbor-tasks/<slug>/`. Save holistic-graded trials in the task's `reference-runs/` and the atomic grades in `rubric-regrades/<run>/`, or carry the relevant job folders from `harbor-jobs/` into the new toolkit too.
2. **Set up the new toolkit.** Refresh your task page, download and unpack the toolkit, then follow Set up the toolkit to configure it.
3. **Copy the task folder.** Copy the entire `harbor-tasks/<slug>/` directory into the new toolkit, keeping its files and grading history together.
4. **Check the grading files.** If the reference runs have already been graded, preserve the task's pinned `tests/` files. If they have not been graded, copy the new toolkit's `task-shared/test.sh` into the task's `tests/` folder before grading them.
5. **Build and verify the environment.** From the new toolkit root inside Authoring, run `bash scripts/build-workspace.sh <slug>`. This rebuilds the workspace from the pinned commit and patch and stages the repository's test commands. Follow Prepare the task environment to verify it.
6. **Review the task against the new scaffold.** Compare it with `harbor-tasks/_task-scaffold/` and add any required rubric sections that are missing. Use Write the holistic rubric to decide what each section needs; do not copy placeholder text into the finished rubric.
7. **Complete or update grading.** For a complete submission, add a missing atomic rubric, grade the saved reference runs under it, and store the grades, using Generate the atomic rubric and its grades. If the atomic rubric and grades already exist, preserve them and check whether any task inputs, rubric guidance, or grader changed. Follow What to rerun after edits for the affected trials and grades.
8. **Refresh detector reports.** Run the detectors included in the new toolkit and review their reports. Address any stale-run or stale-grade findings before submitting.

Older runs without input records

Validation may report that it cannot verify whether older runs match the current task because their input-checksum records are missing. This warning applies to those runs; it does not appear after every toolkit update.

If the prompt, recorded conversation, workspace patch, or pinned commit changed after a run, create a new trial and replace it. Do not treat missing records as evidence that the run matches the current task. Rerunning the affected trials provides results with current input records.

Update the grader

To deliberately move an existing task to the grader supplied by the new toolkit, run this from the toolkit root inside Authoring:

bash

Copy

```
cp task-shared/test.sh harbor-tasks/<slug>/tests/
```

Regrade every reference run with the updated grader, including atomic grading when required. Review the new grades and their agreement using the grading comparison guidance. Use the updated grades consistently across the submission.

Atomic rubrics for older tasks

The September 9 announcement supersedes the September 3 exception. Complete submissions need an atomic rubric and its atomic grades, including tasks created on earlier toolkits or already submitted for feedback. If a task comes back for edits without an atomic rubric, add it alongside the requested edits.

If the task is still on an earlier toolkit, migrate it first to the announced [4ead3c97eb](#) release. Keep [tests/grader-guidance-consolidated.md](#) exactly as it is; the atomic-rubric skill reads it directly. Then generate and verify the atomic rubric, stage it, and regrade every reference run.

For reviewers: return tasks missing the atomic rubric or its regrades for edits. If adding the missing atomic rubric is the only requested change, the return does not count as a review round. Tell the author and count their next submission as the same round.

Recent changes

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

September 16, 2026

Six toolkits are deprecated: [human-essentials](#) , [awbw](#) , [stocks-in-the-future](#) , [community-foundation](#) , [endsideout](#) , and [flaredown](#) . Do not start a new task on one. They stay in the project so tasks already created on them can be finished, revised, and reviewed, so if you have one in progress or returned for edits, keep working in it as normal. Every other toolkit, [casa](#) included, is unaffected. See Toolkits available only for existing tasks.

Anything you create with project tools or tokens is work made for hire. The agreement you signed when you joined makes anything you create for us, or using our tools and tokens, work made for hire owned by DataAnnotation.tech. That includes personal projects built with project model access. See Work made for hire.

Write short rationales, explanations, and feedback yourself. When this project, a side project, or a Slack thread asks for a few sentences of reasoning, they must be your own words. AI may correct your language or translate your own draft; include your original text when it does. See Write rationales and feedback yourself.

September 14, 2026

A new toolkit release, [4ead3c97eb](#) , is available, and every task you have not yet submitted moves onto it. Start every new task on it. Download it from your task page and follow Set up the toolkit. Move every task you are still working on, including one in reviewer feedback, onto this release before you submit it; Toolkit updates lists the steps. We improved the [/write-atomic-rubric](#) skill and the [rubric-form](#) detector to ensure the atomic rubric contains one criterion per scoring-relevant rule. If your task was limited to 24 atomic criteria, and your holistic guidance contained more than 24 scoring-relevant rules, regenerate the atomic rubric with [/write-atomic-rubric](#) after you migrate, then stage it with [npx tsx scripts/stage-atomic-rubric.ts <your-task-slug>](#) . A task that has already been accepted needs nothing. If your task predates the atomic rubric, see Atomic rubrics for older tasks. The release contains the changes below.

New tasks grade each attempt once. A task created on this release produces one grading assessment per trial or regrade, so results arrive sooner and the reward varies more from run to run. To average three assessments instead, set [GRADER_SAMPLES=3](#) as a prefix on [scripts/harbor-run](#) or [scripts/harbor-regrade](#) , or add that line to [.env](#) . A task created on an earlier toolkit keeps the sample count in its own [tests/test.sh](#) , so keep [--verifier-env GRADER_SAMPLES=1](#) in your commands for those tasks. See Grading samples.

An atomic rubric carries as many criteria as its holistic rubric needs. The `/write-atomic-rubric` skill, the `rubric-form` detector, and the review checks no longer limit the number of criteria. Write one criterion for each rule of the holistic rubric that affects the score, and let the count follow from the source. See Atomic rubric criteria.

`harbor-regrade` can grade every reference run in one command. Pass `--all` in place of a single run directory and the script works through every run saved under `reference-runs/`, two at a time. Add `--jobs N` to change how many run at once; each one is a container, so raise it only as far as your machine allows. This is the usual way to regrade after editing a rubric. See Regrade saved runs.

A regrade files itself into `rubric-regrades/<run>/`. An atomic regrade lands in your task's `rubric-regrades/<run>/`, named after the run it graded, without any copying on your part. When a grade is already stored for that run, the new result stays in its job folder under `harbor-jobs/` so you can compare the two first, and `--replace` adopts the new one. See Store the atomic grades.

Leave `allow_internet = true` in `task.toml`. A new task's `task.toml` now says so. Setting it to `false`, or adding `network_mode` or `allowed_hosts`, cuts the grader off from the model and the trial comes back with no reward. See Dependencies and network access.

Writing the atomic rubric no longer marks every earlier detector report stale. Only `rubric-coverage` and `rubric-form` read that file, so the other reports stay fresh, and a staleness warning now names the input the report actually read. See Detectors.

Tasks built from a snapshot start with the full holistic-rubric template. `snapshot-to-task` now scaffolds the same template as a manual task, with every section heading and the current wording for penalties. Task images also no longer report their own dependency setup, such as a changed `Gemfile.lock` or `poetry.lock`, as changes your agent made.

Several repository toolkits received fixes. On the Palolo toolkit, the test suite runs against its own database again, and a bare `pnpm test` needs `--run`. On the human-essentials toolkit, one flaky request spec is skipped so it no longer counts toward your task's test signal, and a task container's development database comes up with the app's demo data loaded. On polyglot toolkits with Python members, `run-app` recovers after a failed setup, tells you when a member is explore-only because its dependencies cannot install, and names the member it picked when you give it no repository. On the ZenBill toolkit, the jest check runs with an explicit heap cap, so it finishes inside the container's memory limit. On the endsideout, stocks-in-the-future, and flaredown toolkits, the graded test checks were corrected so that a red result reflects your agent's work and not the check's own setup. On the zeta-polyglot toolkit, the zeta-wasabi-platform task image no longer enables two integrations the app's own test setup leaves off, setting up a second Python member no longer breaks Poetry, and `run-app` points you at the error on screen when setup fails. On the potion-polyglot toolkit, seven members' task images now build with their dependencies installed, `yarn test` on the potion-app member runs the same suites as the graded jest check, and unzipping the toolkit on macOS or Windows no longer pauses on duplicate files. A task created on an earlier release keeps its own `tests/test-commands.sh` and `environment/Dockerfile`, so the check and image changes reach the tasks you create on this release. See Repositories.

September 11, 2026

You can change dependency versions in [workspace.patch](#) . The trial image installs dependencies after applying your patch, so a package you add, upgrade, downgrade, or remove in a manifest is what the trial actually gets. This makes a dependency upgrade, a version conflict, or an outdated library a supported starting point for a task. Three conditions: every version you name must be one that really exists, because the install resolves against the real registry at build time; update the lockfile in the same patch; and leave the language runtime alone, since the base image fixes it and [environment/Dockerfile](#) is still not yours to edit. Rebuild the workspace and run a trial afterwards to confirm the environment still installs. See Changing dependency versions.

September 10, 2026

Store your atomic grades in [rubric-regrades/<run>/](#) . Grading a reference run under the atomic rubric produces an atomic grade, and every complete submission now carries one per reference run. Each run's grade is stored in [rubric-regrades/](#) inside your task folder as it finishes, one folder per run, named after the run; `--replace` overwrites one you already stored. See Store the atomic grades for the command and Submit for review for the packaging checks.

The existing submit script packages the folder, so no toolkit update is needed. Do not copy atomic grades into [reference-runs/](#) ; that replaces the run's holistic grade, and the detectors flag the package.

This applies to tasks you have not submitted yet and tasks returned for edits. If your task is already out for review, wait for reviewer feedback and add the grades in that revision.

Reviewers check that [rubric-regrades/](#) holds one folder per reference run. If any are missing, return the task and let the author know. As with a missing atomic rubric, that return does not count as a round of review. Apply this check to all reviews.

When storing a grade, set `trial` to the actual completed atomic trial folder on your machine. Its location depends on how grading was launched. Use `reference_run_dir` in the trial's `config.json` to find the original reference run's name for the destination. Existing valid grades do not need to be moved or regenerated. The command replaces the previously stored grade after copying the selected result. The page formerly titled Generate the atomic rubric and regrade is now Generate the atomic rubric and its grades.

September 9, 2026

Older tasks also need atomic rubrics and **regrades**. This supersedes the September 3 exception, including for tasks already submitted for feedback. Migrate an earlier toolkit first, keep the existing

holistic-rubric file, and add the atomic rubric when making requested edits. See Atomic rubrics for older tasks.

Use one grading sample per attempt. Include `--verifier-env GRADER_SAMPLES=1` when running trials or regrading saved runs, with either the holistic or atomic rubric. The report and final score now come from one assessment. You still need at least four reference runs. See Commands and skills and How grading works.

September 8, 2026

Report time at the end of every day; batched time reports are no longer allowed. Log each day's hours before you sign off; if you miss a day, report it at the start of your next session. The section also now spells out what time you can report, including onboarding, Office Hours, and active work while runs execute. See Reporting your time.

September 5, 2026

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.

September 3, 2026 — Two-rubric workflow and Fable 5.1

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. The

original exception for tasks already submitted or iterating on feedback was superseded on September 9: older tasks now also need atomic rubrics and regrades. See [Writing the Holistic Rubric](#), [Generating the Atomic Rubric](#), and [Toolkit Versions & Migration](#).

September 3, 2026 — Codex is the default trial agent

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](#).

September 1, 2026 — Submission Overview

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.

August 31, 2026 — Task-specific checks

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](#).

August 27, 2026 — Source of truth

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.

August 27, 2026 — Penalty targets

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.

August 27, 2026 — Complete submissions take priority

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.

August 25, 2026 — Failure frequency

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.

August 24, 2026 — Review limits

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.

August 24, 2026 — Feedback on the platform

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.

August 10, 2026

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.

August 7, 2026

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.

August 4, 2026

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.

July 27, 2026

Confidentiality. Everything we provide is confidential and stays on your local machine. The Confidentiality section states the policy.

July 5, 2026

Heavy penalties replaced score caps. Dealbreakers are written as heavy penalties rather than caps on the score.

Participation and support

Reporting your time

When to report

Report each day's hours before signing off. Do not save up several days to report in one batch. If you miss a day, report those hours at the start of your next session.

What time counts

Report the time you actively spend on project work, including:

- onboarding and reading the instructions, logbook, and FAQ;
- attending Office Hours;
- exploring repositories, including failed or abandoned attempts;
- writing tasks and rubrics, verifying behavior, reviewing trials and grades, and addressing detector findings; and
- preparing submissions and making revisions after review.

Active project work while trials, grading, or detectors run is reportable. You can explore a repository, read the docs, review results, or work on another task. Time spent only waiting for runs or reviews to finish is not reportable.

Where to report

Use the platform's Report time section for any open task. You do not need to submit the task first, and you can report work on tasks that are blocked or that you do not finish.

For project work without its own Raccoon task, such as the learning exercise or a survey, use the separate off-task time-reporting project. Use the time-off project to let the team know when you will be away.

Good standing

The expected pace typically takes 20–40 hours in most weeks, not every week. There is no weekly cap as long as work quality remains strong.

Milestone	Requirement
First accepted task	Have your first task accepted within your first 50 project hours.
Ongoing standing	Maintain at least two accepted tasks across your most recent 100 hours of Raccoon task work.
Continued access	After four accepted tasks, project access is guaranteed while you remain in good standing.

A task counts as accepted when a reviewer explicitly accepts it, with or without edits. Only task submissions on this project count toward standing; side-project hours do not count toward the 100-hour window.

The Submission Overview shows your hours, milestone progress, queue outlook, and submission history. Display errors do not change your standing. A task awaiting review is not a rejection; once accepted, it counts from its submission date.

Questions and support

Need	Where to go
General project question	#ext-surge-raccoon-questions
Missing or mismatched feedback, a task that has not returned, or escalation after a human review	#ext-surge-raccoon-questions
A message to your reviewer	The Review Logbook on the task
Release announcements	#ext-surge-raccoon-announcements

Read the FAQ before asking a general question. Reply in Slack threads and format code and logs as code blocks. Do not direct-message an admin unless they ask you to.

Discuss task reviews in the Review Logbook on the task.