# For Contributors

We'd love your help! This doc covers how to become a contributor and submit code to the project.

## Where can I start?

We tag issues that are good candidates for those new to the code with [`good first issue`](https://github.com/GoogleChrome/lighthouse/issues?q=is%3Aissue+is%3Aopen+sort%3Aupdated-desc+label%3A%22good+first+issue%22). We recommend you start there!

## Follow the coding style

The `.eslintrc.js` file defines all. We use [JSDoc](http://usejsdoc.org/) with [TypeScript `checkJs`](https://www.typescriptlang.org/docs/handbook/type-checking-javascript-files.html#supported-jsdoc). Annotations are encouraged for all contributions.

## Learn about the architecture

See [Lighthouse Architecture](./docs/architecture.md), our overview and tour of the codebase.

## Contributing a patch

If you have a contribution for our [documentation](https://web.dev/learn/#lighthouse), please submit it in the [web.dev repo](https://github.com/GoogleChrome/web.dev).

1. Submit an issue describing your proposed change.

1. The maintainers will respond to your issue promptly.

1. If your proposed change is accepted, and you haven't already done so, sign a Contributor License Agreement (see details below).

1. Fork the repo, develop and test your code changes.

1. Ensure that your code adheres to the existing style in the sample to which you are contributing.

1. Submit a pull request.

If you've submitted a number of significant patches, feel free to add yourself in a PR to the project's `AUTHORS` [file](https://github.com/GoogleChrome/lighthouse/blob/master/AUTHORS) in the root of the repo to be recognized for your contributions!

## Audit PRs

If proposing a new audit for Lighthouse, see the [new audit proposal guide](./docs/new-audits.md) and open an issue for discussion before starting.

A PR for a new audit or changing an existing audit almost always needs the following:

1. If new, add the audit to the [default config file](lighthouse-core/config/default-config.js) (or, rarely, one of the other config files) so Lighthouse will run it.

1. \*\*Unit tests\*\*: in the matching test file (e.g. tests for `lighthouse-core/audits/my-swell-audit.js` go in `lighthouse-core/test/audits/my-swell-audit-test.js`).

1. \*\*Smoke (end-to-end) tests\*\*: search through the [existing test expectations](lighthouse-cli/test/smokehouse/test-definitions/) to see if there's a logical place to add a check for your change, or (as a last resort) add a new smoke test.

1. Run `yarn update:sample-json` to update the [sample Lighthouse result JSON](lighthouse-core/test/results/sample\_v2.json) kept in the repo for testing. This will also pull any strings needed for localization into the correct files.

### Audit `description` Guidelines

Keep the `description` of an audit as short as possible. When a reference doc for the audit exists on

developers.google.com/web, the `description` should only explain \*why\* the user should care

about the audit, not \*how\* to fix it.

Do:

Serve images that are smaller than the user's viewport to save cellular data and

improve load time. [Learn more](https://developers.google.com/web/tools/lighthouse/audits/oversized-images).

Don't:

Serve images that are smaller than the user's viewport to save cellular data and

improve load time. Consider using responsive images and client hints.

If no reference doc exists yet, then you can use the `description` as a stopgap for explaining

both why the audit is important and how to fix it.

## Gatherer PRs

Gatherers have to interface with the inherently-complex real world and they also run \*while\* loading and/or testing the page, risking interfering with themselves. As a result, gatherers should strive to be as simple as possible and leave as much processing as possible to computed artifacts and audits.

It can be tempting to serialize the entire state of the world into the artifact the gatherer produces. Try to limit the artifact to exactly the information needed for current audits while leaving room for extensibility as needs change in the future.

A PR adding or changing a gatherer almost always needs to include the following:

1. If new, add the gatherer to the [default config file](lighthouse-core/config/default-config.js) (or, rarely, one of the other config files) so Lighthouse will run it.

1. \*\*Unit tests\*\*: gatherer execution often takes place mostly on the browser side, either through protocol functionality or executing javascript in the test page. This makes gatherers difficult to unit test without extensive mocking, ending up mostly exercising the mocks instead of the actual gatherer.

As a result, we mostly rely on smoke testing for gatherers. However, if there are parts of a gatherer that naturally lend themselves to unit testing, the new tests would go in the matching test file (e.g. tests for `lighthouse-core/gather/gatherers/reap.js` go in `lighthouse-core/test/gather/gatherers/reap-test.js`).

1. \*\*Smoke (end-to-end) tests\*\*: search through the [existing test expectations](lighthouse-cli/test/smokehouse/test-definitions/) to see if there's a logical place to add a check for your change, or (as a last resort) add a new smoke test if one is required.

It's most important to get true end-to-end coverage, so be sure that audits that consume the new gatherer output are in the expectations. Artifacts can also have expectations for those intermediate results.

1. \*\*Golden artifacts\*\*: `sample\_v2.json` is generated from a set of artifacts that come from running LH against `dbw\_tester.html`. Those artifacts likely need to be updated after gatherer changes with `yarn update:sample-artifacts`, but limit to just the artifact being altered if possible. For example:

```sh

# update just the ScriptElements artifact

yarn update:sample-artifacts ScriptElements

```

This command works for updating `yarn update:sample-artifacts devtoolsLogs` or `traces` as well, but the resulting `sample\_v2.json` churn may be extensive and you might be better off editing manually.

1. Run `yarn update:sample-json` to update the [sample Lighthouse result JSON](lighthouse-core/test/results/sample\_v2.json) kept in the repo for testing. This will also pull any strings needed for localization into the correct files.

## Protobuf errors

If there is an error in one of the proto tests (`proto-test.js` or `psi-test.js`), you may need to install `protobuf` locally for debugging. See the instructions for installing and running in the [proto readme](proto/README.md).

## Adding Images to a Readme

If you are adding an image to a readme use the absolute path to the image for the specific commit hash where the image was introduced. This requires multiple commits.

1. Make the commit to introduce the image.

1. Get the [absolute path](https://help.github.com/articles/getting-permanent-links-to-files/) to the image with the commit hash e.g. `https://raw.githubusercontent.com/GoogleChrome/lighthouse/e7997b3db01de3553d8cb208a40f3d4fd350195c/assets/example\_dev\_tools.png`

1. Add to readme as an absolute reference to that image.

If you are updating an image that already exists: commit it, then update the readme to point the image with that new commits hash absolute url.

## Pull request titles

We're using [conventional-commit](https://conventionalcommits.org/) for our commit messages. Since all PRs are squashed, we enforce this format for PR titles rather than individual git commits. A [`commitlint` bot](https://github.com/paulirish/commitlintbot) will update the status of your PR based on the title's conformance. The expected format is:

> type(scope): message subject

\* The `type` must be one of: `new\_audit` `core` `tests` `i18n`, `docs` `deps` `report` `cli` `clients` `misc`. (See [`.cz-config`](https://github.com/GoogleChrome/lighthouse/blob/master/.cz-config.js#L13))

\* The `scope` is optional, but recommended. Any string is allowed; it should indicate what the change affects.

\* The `message subject` should be pithy and direct.

The [commitizen CLI](https://github.com/commitizen/cz-cli) can help to construct these commit messages.

## Sign the Contributor License Agreement

We'd love to accept your sample apps and patches! Before we can take them, we have to jump a couple of legal hurdles.

Please fill out either the individual or corporate Contributor License Agreement (CLA).

\* If you are an individual writing original source code and you're sure you own the intellectual property, then you'll need to sign an [individual CLA](https://developers.google.com/open-source/cla/individual).

\* If you work for a company that wants to allow you to contribute your work, then you'll need to sign a [corporate CLA](https://developers.google.com/open-source/cla/corporate).

Follow either of the two links above to access the appropriate CLA and instructions for how to sign and return it. Once we receive it, we'll be able to

accept your pull requests.

## Tracking Errors

We track our errors in the wild with Sentry. In general, do not worry about wrapping your audits or gatherers in try/catch blocks and reporting every error that could possibly occur; `lighthouse-core/runner.js` and `lighthouse-core/gather/gather-runner.js` already catch and report any errors that occur while running a gatherer or audit, including errors fatal to the entire run. However, there are some situations when you might want to explicitly handle an error and report it to Sentry or wrap it to avoid reporting. Generally, you can interact with Sentry simply by requiring the `lighthouse-core/lib/sentry.js` file and call its methods. The module exports a delegate that will correctly handle the error reporting based on the user's opt-in preference and will simply no-op if they haven't so you don't need to check.

#### If you have an expected error that is recoverable but want to track how frequently it happens, \*use Sentry.captureException\*.

```js

const Sentry = require('./lighthouse-core/lib/sentry');

try {

doRiskyThing();

} catch (err) {

Sentry.captureException(err, {

tags: {audit: 'audit-name'},

level: 'warning',

});

doFallbackThing();

}

```

#### If you need to track a code path that doesn't necessarily mean an error occurred, \*use Sentry.captureMessage\*.

NOTE: If the message you're capturing is dynamic/based on user data or you need a stack trace, then create a fake error instead and use `Sentry.captureException` so that the instances will be grouped together in Sentry.

```js

const Sentry = require('./lighthouse-core/lib/sentry');

if (networkRecords.length === 1) {

Sentry.captureMessage('Site only had 1 network request', {level: 'info'});

return null;

} else {

// do your thang

}

```

#### Level Guide

- `info` for events that don't indicate a bug but should be tracked

- `warning` for events that might indicate unexpected behavior but is recoverable

- `error` for events that caused an audit/gatherer failure but were not fatal

- `fatal` for events that caused Lighthouse to exit early and not produce a report

# For Maintainers

The [release guide](./docs/releasing.md).