FOR BUILDERS

THE VIBECHECK SPEC

VibeCheck grades a codebase against an RFP by measuring how closely they resemble each other — not by running your code. This is what that actually means for how you write, name, and document what you build, so your submission maps as cleanly as possible to what was asked for.

Applies to: **any repo submitted to VibeCheck** Audience: **builders & submitters** Status: **living document**

TL;DR

- VibeCheck compares **text** — your code's names, comments, and structure — against the RFP's requirement text. It never executes anything.
- Your **README counts as code**. It's chunked and matched exactly like a source file, and often makes the strongest match of all.
- Each requirement is matched to **one** function or class, not scattered fragments — give each capability a single, well-named home.
- Name things the way the **RFP talks about the problem**, not the way your codebase happens to think about it internally.
- One missed **mandatory** requirement grades the whole run a Fail — the score itself isn't capped or altered, only the grade.
- An unmatched **optional** requirement costs nothing. It only ever helps your score if matched — never hurts it if skipped.

01 How VibeCheck actually reads your repo

Five stages, all automatic, all happening before a human ever opens your repo. Knowing the shape of this pipeline is the fastest way to understand why some submissions score higher than others that arguably "did more."

01
Filter
Strips vendor dirs, build output, binaries, lockfiles, and anything your `.gitignore` already excludes.

02
Chunk
Splits what's left at real function/class boundaries — not fixed line windows.

03
Embed
Every chunk and every RFP requirement is embedded into the same vector space by a code-aware model.

04
Match
Each requirement is paired with its single best chunk — no chunk gets reused across two requirements.

05
Score
Matched requirements contribute their similarity, weighted by importance, to a 0–100 score — optional ones only ever add to it.

The important part: this is **semantic text comparison**, not compilation and not test execution. VibeCheck is judging how closely your code's actual written content — names, comments,

structure, docs — resembles what the requirement describes. It rewards code that reads clearly, in roughly the same terms the RFP uses, over code that merely happens to work.

02 What gets read, what doesn't

The filter pass runs before anything else. If a file doesn't survive it, it's invisible to matching entirely — it can't help your score, but it also can't hurt it.

✓ INCLUDED

- All your source code, in any language
- `README.md` and any other docs
- Config files, scripts, anything text-based
- Comments and docstrings inside your code

✗ EXCLUDED

- `node_modules`, `vendor`, `.venv`, `dist`, `build`, `.git`
- Images, audio, video, fonts, archives, binaries
- Lockfiles (`package-lock.json`, `Cargo.lock`, etc.)
- Anything your own `.gitignore` already excludes
- Any single file over 500KB

That first bullet under "included" is the one people miss: **your README is not a courtesy for human reviewers — it's data VibeCheck actually embeds and matches on**, the same as any source file. A clear README that plainly states what you built can out-match code that does the same thing but never says so anywhere.

03 How your code gets chunked

VibeCheck doesn't compare your whole repo against a requirement, or even a whole file — it breaks everything into small pieces first, and matches each requirement to exactly one of them.

- Structured languages (JS/TS, Python, Go, Java, C#, Ruby, PHP, Rust, Kotlin, Swift, C/C++) are split at real declaration boundaries — one chunk per function, class, interface, struct, or enum.
- Everything not inside one of those — top-level imports, constants, loose script logic — gets bundled into its own chunk, in windows of up to 200 lines.
- Non-code text (READMEs, docs, config) has no declarations to split on, so it's windowed the same way: up to 200 lines per chunk.

> **Practical effect**
>
> If a requirement's implementation is spread across a dozen small, loosely-related snippets with no single piece that clearly does the job, none of those fragments individually looks like a strong match — the signal gets diluted across all of them. A single, well-named function or class that clearly owns that capability is chunked as one coherent unit and matches far better.

04 Writing code that matches well

None of this changes what you should build — only how clearly what you already built reads back to something comparing text. Four habits do most of the work.

1. Name things the way the RFP talks about the problem

Matching is semantic, not literal string search — but a name that echoes the RFP's own vocabulary still gives the embedding model a much stronger, more direct signal than a name that's accurate only to someone who already knows your codebase.

WEAKER MATCH

```
function f(o) {
  return fetch(o.url, { timeout: 3000 })
    .then(() => true)
    .catch(() => false);
}
```

No name, no comment, no domain language — the model has almost nothing to compare against the requirement text.

STRONGER MATCH

```
// Checks whether the machine currently
// has working internet connectivity.
function checkInternetConnectivity(opts) {
  return fetch(opts.url, { timeout: 3000 })
    .then(() => true)
    .catch(() => false);
}
```

Same logic. The name and comment now say, in plain language, exactly what the requirement asks for.

2. Write comments that say what and why, not just how

A comment describing mechanics ("loop over the array") adds little a requirement would ever be phrased like. A comment describing intent ("retries with backoff so a flaky network doesn't fail the whole check") speaks the same register an RFP requirement does.

3. Give each requirement one clear home

One well-scoped function or module per capability beats the same capability's logic spread across several unrelated files with nothing tying them together — see the chunking section above.

4. Say what you built, in your README

Because docs are chunked and embedded exactly like code, a README section that plainly states "This project checks internet connectivity via DNS lookup with a configurable timeout and fallback hosts" is itself a real, standalone match candidate — sometimes the strongest one in the whole repo. Don't treat the README as an afterthought written after the scoring matters.

05 What doesn't help

Worth being direct about, since it cuts both ways.

Stuffing keywords doesn't reliably help

Matching runs on meaning, not literal word overlap — piling RFP phrases into a comment above code that doesn't actually do that thing is more likely to read as noise than as a match.

More code isn't a better score

Volume unrelated to any requirement doesn't sit quietly — it surfaces explicitly, as unexplained code, in the report anyone reading your results will see. A smaller, focused repo where everything ties back to something asked for reads better than a large one where most of it doesn't.

Also worth knowing

VibeCheck measures resemblance, not verified correctness. It doesn't execute, test, or confirm your code actually works — a well-named, well-documented function that's broken or empty can still score as a match. That's exactly why VibeCheck is framed as a triage signal for a human reviewer, not a pass/fail compliance check on its own. Writing clearly for VibeCheck and writing correctly are two different jobs — do both.

06 If you're also writing the RFP

Everything above assumes the RFP itself extracts into clean, gradeable requirements. If you control that document too, a few habits make the extraction more accurate on your behalf.

- State one discrete ask per requirement. A sentence bundling two asks together ("the system shall support X and provide Y") tends to get split, but a clean split of your own removes any ambiguity about which part matters.
- Use `shall` / `must` language for anything genuinely non-negotiable, and `should` / `may` for anything that isn't — that's the actual signal used to infer which requirements are mandatory and which are optional.
- Keep submission logistics (page limits, contact info, legal boilerplate) out of the functional-requirements section — that kind of text is deliberately filtered out during extraction, so mixing it in just adds noise to skim past.

07 Technical reference

PARAMETER	VALUE
Embedding model	`voyage-code-3`, one shared vector space for code and requirement text
Chunking unit	One chunk per function/class/interface/struct/enum; leftover code and non-code text windowed at ≤200 lines
Match threshold (default)	`0.75` similarity — adjustable per run without re-embedding anything
Requirement assignment	Optimal one-to-one (Hungarian algorithm) — a chunk can't be claimed by two requirements
Max file size read	`500KB` per file
Max chunks per repo	`2,000` — larger repos are sampled, favoring larger chunks
Mandatory-miss consequence	The run is graded **Fail**, regardless of the number — the score itself is no longer capped or altered
Unmatched optional requirement	No effect on the score at all — only a match adds its weighted share; skipping one costs nothing
Excluded by default	Vendor/build directories, binaries, media, archives, lockfiles, anything gitignored

This document describes how VibeCheck's matching pipeline actually behaves today and will be updated if that changes. It's guidance for writing clearly, not a scoring guarantee — read the disclaimer on every VibeCheck report for what the score does and doesn't verify.