the unofficial

AI Code Optimization Checklist

⚒ AI Craftspeople Guild • Where AI-assisted codebases waste tokens — and what to reclaim first

The bottom line

Waste is not neutral. Every unnecessary token is a tax on every future interaction with your codebase. If these are true, the tax is low.

- [ ] An AI agent loads **only what it needs** to do a task — not the whole haystack
- [ ] The codebase **resists** accumulating new waste, by structure and habit
- [ ] Reading any one part doesn't drag in piles of **dead or stale context**
- [ ] Token cost per AI interaction is **trending down**, not creeping up
- [ ] Optimized for **both** human readability and AI maintainability

Highest-leverage waste

The biggest token savings for the least effort. Start here — these compound the fastest.

- [ ] **Dead code elimination**
  - [ ] Unused functions & unreachable branches removed
  - [ ] Commented-out blocks & orphaned modules cleared
  - [ ] No AI "scaffolding" left loaded into context
- [ ] **File & module size management**
  - [ ] No monolithic, ever-growing files (the 3,000-line file)
  - [ ] Oversized files split into focused, right-sized units
  - [ ] Minimal context required for any individual task
- [ ] **Duplicate code consolidation**
  - [ ] Near-duplicate functions merged
  - [ ] Parallel implementations of one idea unified
  - [ ] Consolidated into shared, well-named abstractions
- [ ] **Comment & documentation hygiene**
  - [ ] Comments that merely restate the code removed
  - [ ] Obsolete decisions & stale behavior notes cleared
  - [ ] Genuinely valuable documentation preserved

Standard targets

The routine waste of AI-assisted codebases. Usually worth doing on any project of size.

- [ ] **Naming efficiency**
  - [ ] Excessively long folder / file / variable names tightened
  - [ ] Names are concise **and** unambiguous — not cryptic
  - [ ] No token cost without added clarity
- [ ] **Directory structure flattening**
  - [ ] Deep, baroque trees flattened
  - [ ] Path overhead on file references reduced
  - [ ] Layout is semantically clear & discoverable
- [ ] **Dependency trimming**
  - [ ] Packages added for what stdlib could do — removed
  - [ ] Dependency surface shrunk
  - [ ] Less context for agents & humans to reason about
- [ ] **Configuration file compaction**
  - [ ] Spelled-out defaults removed; only intentional settings kept
  - [ ] Verbose, obsolete config comments cleared
  - [ ] Intent immediately legible at a glance
- [ ] **Test suite efficiency**
  - [ ] Repetitive setup & duplicated assertions reduced
  - [ ] Universally-loaded, rarely-used fixtures trimmed
  - [ ] Faster CI; clear failure signals, not buried ones

Context-dependent

Only if your system embeds AI agents. High payoff where it applies — these run on every call.

- [ ] **Prompt & system-message optimization**
  - [ ] Verbose / redundant system prompts tightened
  - [ ] Per-call token overhead reduced
  - [ ] Behavioral clarity preserved or improved
  - [ ] Poorly-structured prompts that degrade output fixed

Positive indicators

Signs your codebase resists waste — the structures and habits worth protecting.

- [ ] Agents **reorganize** rather than just append
- [ ] New files arrive right-sized, not destined to bloat
- [ ] Duplication gets **noticed and merged**, not re-solved
- [ ] AI suggestions are getting **better** as the codebase gets leaner
- [ ] The team treats token cost as a **real cost**, not an abstraction
- [ ] Cleanup is continuous & cheap — not a dreaded annual purge

The compounding return

- [ ] Optimization isn't a one-time cut. It pays back on **every future AI interaction** — cheaper maintenance, better suggestions, slower waste accumulation. The earlier you do it, the bigger the return. But it's **never too late.**

Token = the unit an AI agent reads & is billed on • **Scaffolding** = throwaway code an agent leaves behind • **Context window** = everything the agent loads to do a task • **stdlib** = a language's standard library

This checklist is a **discussion tool, not a compliance scorecard.** Inspired in form by Henrik Kniberg's "unofficial Scrum Checklist." Optimization areas inspired by a chat with Thomas Frumkin.

⚒ AI Craftspeople Guild

AI Code Optimization Checklist — How to use it

What is this? Who is it for?
A simple tool to find where an AI-assisted codebase is quietly wasting tokens — money, latency, and worse AI output — and to decide what to reclaim first. AI agents build fast, but they append rather than reorganize, comment compulsively, leave scaffolding behind, and re-solve problems they've already solved. The waste accumulates like water damage: slowly, invisibly, then all at once. **These aren't rules. They are guidelines.** A tiny project may not need directory flattening; a system with no embedded agents skips the prompt section entirely. Fine — as long as you skipped it *on purpose*. That judgment is the whole point.

How do I use it?

Use it as a **discussion tool** — ideally as a recurring cleanup pass, not a one-time event.

> **Maya:** "Our AI suggestions have gotten worse lately, and our token bill is climbing. Where's the waste?"
>
> **Dev:** "Honestly? We've got a couple of 3,000-line files the agent keeps appending to. It reloads all of it for every small change."
>
> **Maya:** "That's 'file & module size' — top of the high-leverage column. Big saving, low effort. Let's start there."
>
> **Dev:** "And there's dead scaffolding everywhere from the last sprint. Plus the agent re-wrote the same date helper in three places."
>
> **Maya:** "Dead code and duplication — also high-leverage. One afternoon on these three and the next month gets cheaper."

That's the right use: find the waste with the biggest return, reclaim it, and let the saving compound.

Why AI-built code accumulates waste

It's structural, not careless. Agents append rather than reorganize. They comment compulsively. They leave scaffolding behind and solve the same problem twice without noticing. None of it shows up in a demo — but every unnecessary token is loaded again on every future interaction, for the life of the project.

How do I *NOT* use it?

Don't turn token-counting into a blunt instrument.

> **Big Boss:** "Shortest names possible, everywhere. I want maximum token savings by Friday."
>
> **Dev:** "We renamed everything to one and two letters. Token count's down 4%."
>
> **Big Boss:** "More! Strip every comment too. Comments are waste!"
>
> **Maya:** "Now nobody — human or agent — can read it, and we deleted the three comments that actually explained why. We traded a tiny saving for real confusion."
>
> **Big Boss:** "The checklist says trim, so trim. No exceptions."

Optimization removes waste that costs tokens **without adding value.** It never sacrifices clarity for a smaller number. Concise **and** unambiguous — both, or you've optimized the wrong thing.

Is this an official standard?

No. It reflects the Guild's craft-grounded view of where AI-assisted codebases waste tokens and what's worth reclaiming. Argue with it, adapt it, experiment. If it surfaces the right cleanup before the waste compounds, it's done its job.

The one principle

Waste is not neutral. Every unnecessary token is a tax on every future interaction with your codebase.

Inspired in form by Henrik Kniberg's "the unofficial Scrum Checklist" (crisp.se/scrum/checklist). Optimization areas inspired by a chat with Thomas Frumkin.