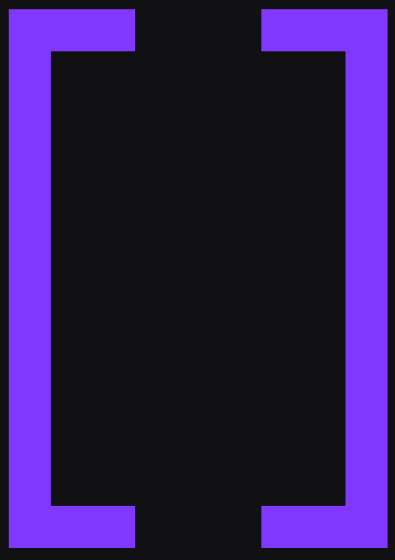
proto[code]

CODE REVIEW

A Reviewer's Field Guide to Slop

Eight kinds of slop in agent code, how to spot each one in a diff & the one-line rule that keeps it out.



Matt Hensley · November 2026

proto[code]

CONTENTS

What's in Here

01 **Slop Is a Review Problem**
What slop is · Where the cost lands

02 **Three Kinds of Slop**
Bloated · Duplicated · Half-finished

03 **The Eight Patterns**
The tell in the diff · The rule that keeps it out

04 **One Diff, Seven Patterns**
An annotated example

05 **Catch It Before Review**
Instructions · Tooling · Review

06 **Reviewing for Slop**
The outside-in scan · An afternoon audit

07 **Resources**
Guidelines & tools

01

Slop Is a Review Problem

Slop is code that works, or looks like it works, & costs more to read & maintain than the task called for.

An agent can write a diff faster than you can read the first file. The writing got cheap. The reading didn't, & every line the agent adds is one your team reviews today & maintains for as long as it exists.

My bar for agent code is the smallest change that fully does the job, written the way the rest of the codebase already writes it. Google's code review guidelines draw the same line around over-engineering:

> *Encourage developers to solve the problem they know needs to be solved now, not the problem that the developer speculates might need to be solved in the future.*
>
> Google Engineering Practices, What to Look For in a Code Review

FIGURE 1 **What the task needed & what the agent sent**

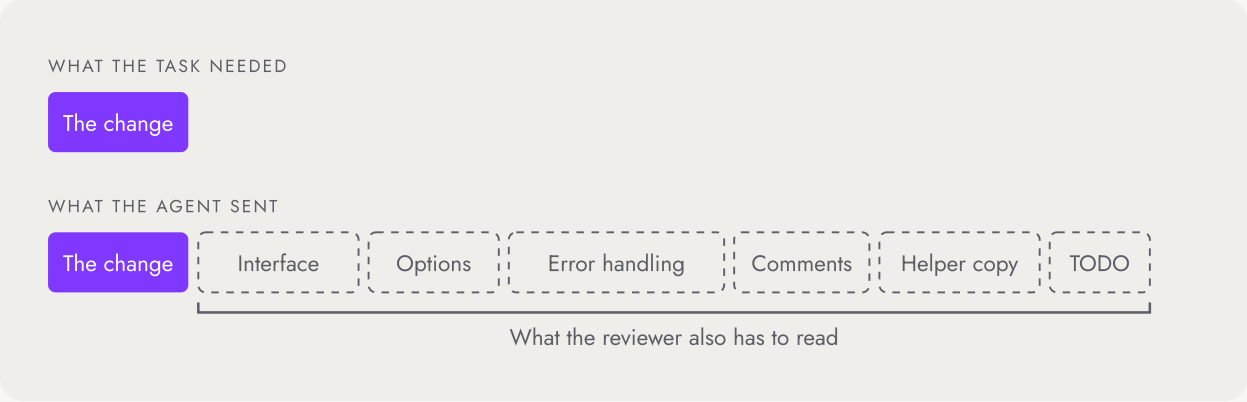


Figure 1. An illustrative diff for a one-line task. The change you asked for is the small part, & the reviewer reads all of it.

02

Three Kinds of Slop

Slop comes in three kinds: code the task didn't need, code that already exists & code that isn't finished.

FIGURE 2 **Eight patterns in three kinds**

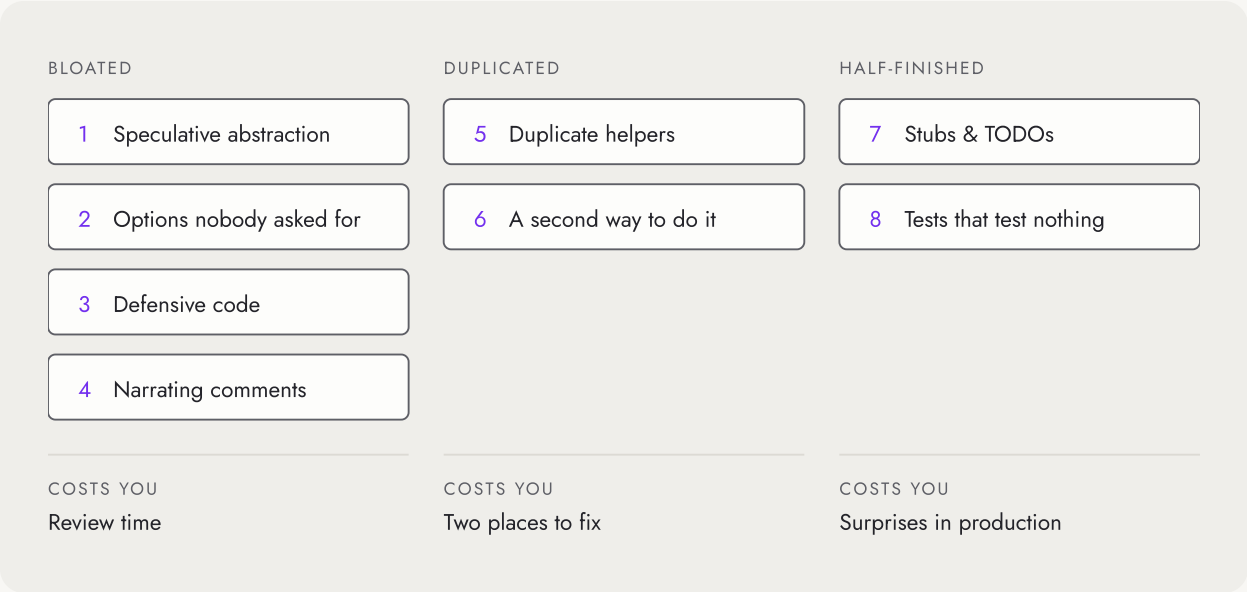


Figure 2. The eight patterns by kind. The numbers stay the same throughout.

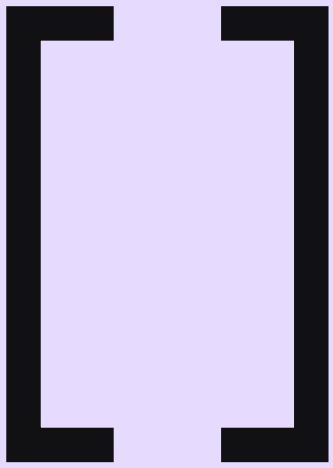
Each kind costs you something different. Bloat costs review time. Duplication leaves two places to fix, & the next bug fix lands in only one of them. Half-finished code passes a skim & you find out in production.

Bloat is the easiest to spot & the easiest to wave through. That's why it's worth pushing back on first. A stub or a hollow test is much easier to miss in 300 lines than in 30.

03

The Eight Patterns

Each pattern has a tell you can spot in the diff & a one-line rule you can hand to the agent before it writes anything.



03

The Eight Patterns

Rows are patterns. The middle column is the tell in the diff, & the last is a rule short enough to paste into your agent's instructions.

BLOATED

PATTERN	SPOT IT IN THE DIFF	THE RULE
1 **Speculative abstraction**	A new interface, base class or factory with exactly one implementation. A Manager or Provider wrapped around a single case.	Add the abstraction when the second case shows up.
2 **Options nobody asked for**	New parameters every caller leaves at the default. New config keys. A flag for behavior the task never mentioned.	Build what the task asks for & nothing it might ask for later.
3 **Defensive code**	Null checks on values the types already guarantee. A try/catch that logs & rethrows. The same validation at every layer.	Validate at the boundary & trust the types inside it.
4 **Narrating comments**	A comment above each line saying what the line does: `// Save the order.`	Comment the why. The code already says what.

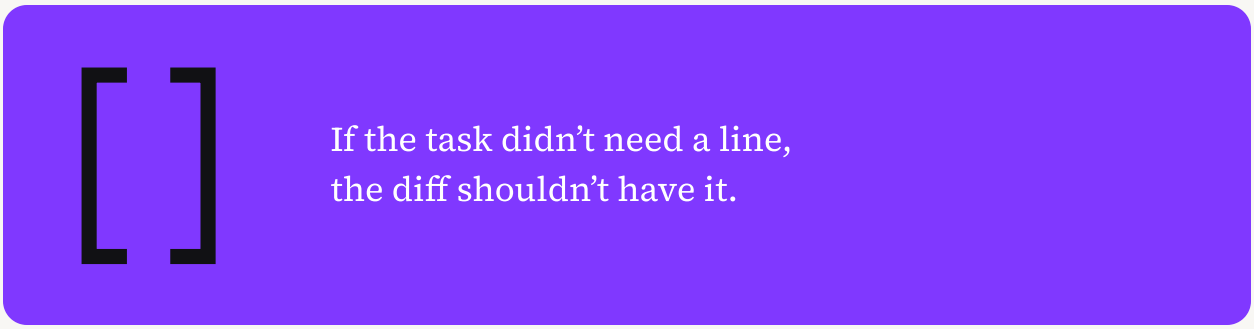
DUPLICATED

PATTERN	SPOT IT IN THE DIFF	THE RULE
5 **Duplicate helpers**	A new private method with a generic name in a feature file: `FormatDate`, `IsValidEmail`, `RoundToCents`.	Search the codebase before you write a helper.
6 **A second way to do it**	A different error handling style, a new library for a job an existing one does, a folder shape no other feature uses.	Follow the pattern the codebase already uses. When there isn't one, ask.

HALF-FINISHED

PATTERN	SPOT IT IN THE DIFF	THE RULE
7 **Stubs & TODOs**	TODO comments, `NotImplementedException`, hard-coded return values, sample data left in place, new code nothing calls yet.	Finish every path, or list the gap in the pull request.
8 **Tests that test nothing**	Assertions that only check a mock was called. Expected values edited to match new output. Skipped tests.	A test has to fail when the behavior it covers breaks.

Keep the rules this short. A reviewer can quote one in a comment, & an agent can follow one without guessing what you meant.



04

One Diff, Seven Patterns

Here is the kind of diff an agent sends back for a one-line task: give loyalty members 10% off the order total.

FIGURE 3 **The diff, annotated**

```
+  public interface IDiscountService
+  {
+      decimal Apply(Order order, decimal rate = 0.10m);  (2)
+  }
+
+  public class DiscountService(ILogger<DiscountService> logger) : IDiscountService  (6)
+  {
+      public decimal Apply(Order order, decimal rate = 0.10m)
+      {
+          try
+          {
+              // Apply the rate if the customer is a loyalty member  (4)
+              var discount = order.IsLoyaltyMember ? rate : 0m;
+              return RoundToCents(order.Subtotal * (1 - discount));
+          }
+          catch (Exception ex)
+          {
+              logger.LogError(ex, "Discount failed");
+              throw;
+          }
+      }
+
+      private static decimal RoundToCents(decimal value) =>
+          Math.Round(value, 2, MidpointRounding.AwayFromZero);
+
+      // TODO: support tiered loyalty levels  (7)
+  }
```

(1) (3) (5)

Figure 3. 27 lines for a one-line task, & nothing calls the new class yet. Each number is one of the eight patterns.

FIGURE 4 **What the task needed**

```
     public decimal Total(Order order) =>
-        Money.Round(order.Subtotal);
+        Money.Round(order.Subtotal * (order.IsLoyaltyMember ? 0.9m : 1m));
```

Figure 4. One changed line, in the method that already owns the total, using the rounding helper the codebase already has.

05

Catch It Before Review

Review is the most expensive place to catch slop. Stop each pattern at the earliest stage that can catch it.

FIGURE 5 **Three places to stop a pattern**

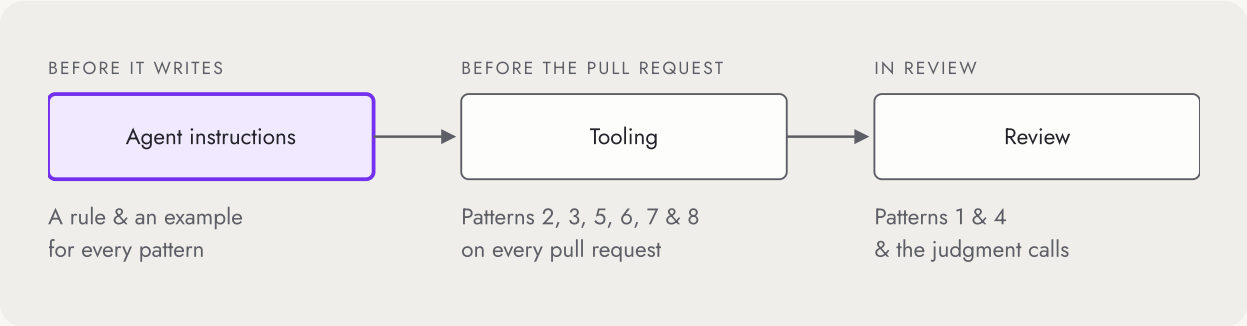


Figure 5. Each stage catches what the one before it missed. Review gets what's left.

I'd rather an agent read a rule once than write the same review comment every week. A rule works best with an example from your own codebase beside it, so the agent copies the right shape instead of guessing at it.

PATTERN	CATCH IT WITH
1 **Speculative abstraction**	A rule in the agent's instructions, with an example of when your codebase does add an interface. Review catches the rest.
2 **Options nobody asked for**	Unused parameter analyzers catch the options nothing reads: IDE0060 in .NET, no-unused-vars in ESLint. Review catches the ones no caller sets.
3 **Defensive code**	Strict null checking, plus a linter rule for conditions that can never be true, like typescript-eslint's no-unnecessary-condition.
4 **Narrating comments**	A rule in the agent's instructions, then review. Telling a useful comment from a narrating one takes judgment.

PATTERN	CATCH IT WITH
5 **Duplicate helpers**	Copy & paste detection (jscpd, PMD's CPD) for near copies. A "search before you write" rule for helpers that do the same job with different code.
6 **A second way to do it**	Architecture tests (ArchUnit, NetArchTest) for layers, dependencies & naming. An example of the existing pattern in the agent's instructions.
7 **Stubs & TODOs**	A CI check that fails when a pull request adds TODO, FIXME or `NotImplementedException`.
8 **Tests that test nothing**	Mutation testing (Stryker) breaks your code on purpose & shows which tests still pass.

06

Reviewing for Slop

Most slop shows up before you read a line of logic, if you read the diff from the outside in.

FIGURE 6 **The outside-in scan**

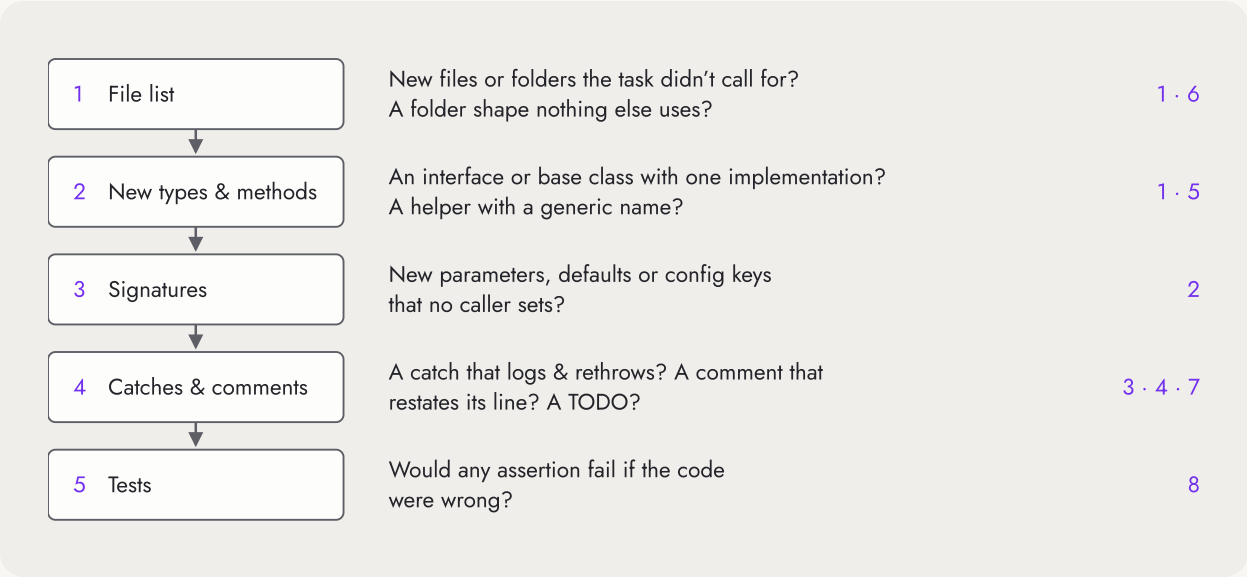


Figure 6. Work from the file list inward. The numbers on the right are the patterns each step catches.

When the scan turns up a pattern, send the pull request back with the rule quoted. Cleaning it up yourself fixes one pull request. Adding the rule to the agent's instructions fixes the next one too.

An Afternoon Audit

You can find out which patterns your agents fall into most in an afternoon. Do it with the people who review the most agent code. They already know which patterns they're tired of.

1. Pick 10 recent pull requests where an agent wrote most of the code.
2. Scan each one outside in & tally the patterns you find.
3. Next to each pull request, note how many lines the task needed & how many it added. A rough guess is fine.
4. Take your top three patterns. Add their rules to the instructions your agents read, each with one example of the right way from your own code.
5. Add a tooling check for any of the three that a tool can catch.
6. Two weeks later, run the same audit on the next 10 & compare.

FIGURE 7 **A tally sheet**

PATTERN	PR 1	PR 2	PR 3	PR 4	PR 5	PR 6	PR 7	PR 8	PR 9	PR 10	TOTAL
1 Speculative abstraction											
2 Options nobody asked for											
3 Defensive code											
4 Narrating comments											
5 Duplicate helpers											
6 A second way to do it											
7 Stubs & TODOs											
8 Tests that test nothing											
Lines needed / added											

Figure 7. One column per pull request. The totals tell you which rules to write first.

The writing got cheap. Keep the reading cheap too: put the rules where the agent reads them before it starts, & save review for the calls only a person can make.

07

Resources

The guidelines & tools behind these patterns, if you want to go deeper.

What to Look For in a Code Review: Google's guidance on complexity & over-engineering

google.github.io/eng-practices/review/reviewer/looking-for.html

AGENTS.md: a shared format for the instructions coding agents read

agents.md

ArchUnit: architecture rules as unit tests for Java

archunit.org

jscpd: copy & paste detection across languages

github.com/kucherenko/jscpd

Yagni: Martin Fowler on building features before you need them

martinfowler.com/bliki/Yagni.html

Stryker: mutation testing for .NET, JavaScript & Scala

stryker-mutator.io

NetArchTest: architecture rules as unit tests for .NET

github.com/BenMorris/NetArchTest

no-unnecessary-condition: typescript-eslint's rule for checks that can never fire

typescript-eslint.io/rules/no-unnecessary-condition

proto[code]

prototypd.dev/code