



Implementing AI in engineering work

Applying AI to engineering knowledge,
documentation, review, and design support with
clear technical accountability.

WORKS WITH THE ASSISTANTS TEAMS ALREADY USE



EDITION 32 / EMBER & CLAY

EXECUTIVE WORKING EDITION - 08 OCT 2026

32

AT A GLANCE

Applying AI to engineering knowledge, documentation, review, and design support with clear technical accountability.



AI assistants can speed up code, tests, documentation, requirements analysis and design review, and they can produce plausible mistakes just as quickly. This playbook is for software and general engineering teams who want the benefit without lowering the bar. It covers where assistance helps, how to write good requests, what to verify, which security cautions apply, how to measure real value, and how to roll out in a way that keeps engineers accountable for what ships.

WHO IT IS FOR

Engineering managers, tech leads, staff engineers, QA and security leads, and engineers in software, hardware, civil, mechanical and systems teams who use or plan to use AI assistants.

TIME TO USE

75 minutes to read; a team workshop of two hours to set standards



What you will have at the end

- A team standard for what AI may be used for and what must be verified
- A set of reusable request templates for code, tests, review and documentation
- A verification routine engineers can apply in minutes
- A measurement plan that looks at quality and rework, not only speed

Contents

01	What AI changes in engineering, and what it does not	03
02	Where AI helps engineering and where it does not	04
03	How assistants produce output, and why that matters for checking	06
04	Team standards, roles and request templates	08
05	Security, privacy, intellectual property and review controls	10
06	Measuring quality and value without fooling yourself	12
07	A rollout plan for an engineering team	14
08	Common pitfalls and an illustrative scenario	16
09	Questions to ask when choosing and configuring tools	17
10	A first quarter that produces evidence	19
11	Terms used in this publication	20



EXECUTIVE BRIEF

What AI changes in engineering, and what it does not

01



AI shortens the path from intent to a first draft. It does not shorten the path from a draft to something trustworthy. That second path is still engineering.

Engineering work involves a lot of drafting: boilerplate code, unit tests, migration scripts, API documentation, requirement summaries, design notes, review comments, runbooks and incident write-ups. Language models are good at producing a plausible first version of all of these. Used well, they remove blank-page time and tedious repetition, and they help engineers explore options faster. Used carelessly, they add subtle defects, invented references and security weaknesses that are harder to spot because the output looks tidy.



The principle that keeps teams safe is accountability. The engineer who commits or signs off a piece of work owns it, whatever tool helped produce it. An assistant has no responsibility for the production outage, the failed audit or the structural calculation. That is why the whole approach in this playbook is built around verification habits rather than around trusting a better model.



- Use assistance where drafts are cheap to check and expensive to write
- Never merge, deploy or sign off what you cannot explain
- Keep secrets, customer data and proprietary material out of tools that are not approved for them
- Measure rework, defects and review load alongside speed
- Write down team standards so that the tool is used consistently, not individually

RULE OF THUMB

If you could not review a human colleague's version of this change, you cannot accept the AI version of it either. Assistance changes who writes the first draft, not who is responsible.



This playbook is hands-on. It includes request templates, review checklists and a rollout plan you can adapt. It assumes a team that already has version control, code review and automated tests; without those, AI mostly accelerates the production of unreviewed change.

WORKING QUESTION

Where in your current workflow does a good first draft save the most time, and how would you know that the draft was good?

WHERE IT FITS

Where AI helps engineering and where it does not

02

Assistance is strongest on well-defined, checkable tasks and weakest where the right answer depends on context the tool cannot see.

TASK	FIT	WHY
Boilerplate, scaffolding, config	Strong	Pattern-heavy, easy to run and review
Unit tests for existing functions	Strong with review	Fast to produce; check they test behavior and not only the current code
Explaining unfamiliar code	Strong	Speeds onboarding; verify against the code and run it
Documentation drafts and changelogs	Strong	Engineer corrects facts; saves writing time
Refactoring with test coverage	Moderate to strong	Tests provide a safety net
Debugging with logs and stack traces	Moderate	Good hypotheses, but needs real reproduction

TASK	FIT	WHY
Architecture and design decisions	Moderate	Useful as a sounding board, not as the decider
Security-sensitive code (auth, crypto, input handling)	Weak without expert review	Subtle flaws, plausible but wrong patterns
Safety-critical or regulated design calculations	Not a replacement	Qualified engineers verify and sign off



Beyond software

General engineering teams in mechanical, civil, electrical or systems work can use the same pattern on documents rather than code: summarizing standards and specifications supplied to the tool, drafting test plans and inspection checklists, comparing requirement versions, preparing design review packs, writing calculation narratives, and turning meeting notes into action lists. Numerical results, load cases, tolerances and compliance conclusions must be computed and checked with your validated tools and by qualified engineers. A language model is not a calculation engine and should not be treated as one.



TEAM TYPE	GOOD EARLY USES	KEEP HUMAN-OWNED
Application software	Tests, docs, code explanation, small refactors	Architecture, security design, release decisions
Platform and infrastructure	Runbooks, config drafts, incident summaries	Production changes, access policy
Data engineering	SQL drafts, pipeline docs, data dictionary	Definitions of metrics, data quality rules
Hardware and mechanical	Requirement summaries, test plan drafts, review notes	Calculations, safety cases, sign-off
Civil and infrastructure	Report drafts, specification comparisons	Design checks, certification, stamped work



COMMON MISTAKE

Judging the tool by how it handles a greenfield demo. Real value and real risk show up in your existing codebase, with your conventions, dependencies and constraints.



WORKING QUESTION

Which three tasks in your last sprint were pattern-heavy and easy to verify, and which were judgment-heavy?

HOW IT WORKS

How assistants produce output,
and why that matters for checking

03

Models predict plausible text from the context you provide. Knowing this explains both their usefulness and their typical failures.

A coding or writing assistant takes your request plus whatever context it can see, such as open files, selected code, documentation you paste or a repository it can search. It then generates the most likely helpful continuation. It does not run your code unless it is connected to tools that do, and it does not know your architecture beyond what it is shown. When context is thin, it fills gaps with common patterns, which may not be yours.

- Invented APIs: calls to functions, flags or packages that do not exist, or that exist in a different version
- Plausible but wrong logic: edge cases, off-by-one errors, concurrency and time zone mistakes
- Outdated patterns: deprecated methods or insecure examples common in older public code
- Tests that mirror the code: assertions that simply restate the current behavior, including its bugs
- Overconfident explanations: a fluent account of why something works that is not true
- Context loss: forgetting earlier constraints in a long session

Agentic modes go further: the tool plans several steps, edits files, runs commands and tests, and iterates. This can be productive on bounded tasks with good tests, and it raises the stakes on permissions. An agent with write access to a repository or a shell can do real damage quickly. Decide what it may touch, run it in a sandbox or on a branch, and review the diff as

you would review a stranger's pull request.

01 Give context

Provide the relevant files, interfaces, constraints and the definition of done. Less guessing means fewer inventions.



02 Ask for a small step

Request one function, one test file or one section. Large asks hide errors.

03 Run it

Compile, execute tests, lint and try the failure cases. Reading alone is not verification.

04 Review the diff

Check what changed, including files you did not expect. Look for removed checks and silent behavior changes.

05 Explain it back

If you cannot explain why the change is correct, ask the tool to explain, then confirm against the code and documentation.



WORKING QUESTION

What context does your team routinely forget to give an assistant, and what does it then guess?



OPERATING MODEL

Team standards, roles and request templates

04

Consistency matters more than any single tool. Agree how the team works, then give people templates that make good requests easy.

ROLE	RESPONSIBILITY	DECISION THEY KEEP
Engineering manager	Sets expectations, protects review time, tracks outcomes	Which uses are encouraged or restricted
Tech lead	Owens the team standard and templates	What good looks like for the codebase
Author engineer	Verifies and owns every change	Whether to submit it
Reviewer	Reviews AI-assisted changes with the same rigor	Approve, request changes or reject
Security lead	Defines data and code rules, reviews risky patterns	What may never be shared or generated
QA lead	Keeps tests meaningful and coverage honest	What evidence is enough for release

Request templates

A good request states the task, the context, the constraints and the form of the answer. Teams that keep a shared library of templates get more consistent results and spend less time re-explaining. Adapt the templates below and keep them in your repository.

- Code change: 'Task: add X to module Y. Context: [paste relevant interfaces]. Constraints: follow our error handling pattern, no new dependencies, keep public API unchanged. Output: a diff and a list of assumptions you made.'
- Tests: 'Write tests for function F. Cover normal cases, empty input, boundary values and the failure modes in this list. Do not assert on implementation details. State any behavior you think is a bug.'

- Review: 'Review this diff for correctness, security issues, missing error handling and unclear naming. List concerns by severity and quote the line. Say what you could not assess.'
- Documentation: 'Draft a README section for this module from the code below. Mark any statement you inferred and could not confirm from the code.'
- Debugging: 'Given this stack trace and code, list three plausible causes, what evidence would confirm each, and the quickest check for each.'
- Requirements: 'Summarize this specification into numbered, testable requirements. Flag ambiguities and contradictions instead of resolving them.'

ASK FOR ASSUMPTIONS

The most useful line in any template is 'list the assumptions you made and what you could not verify.' It turns hidden guesses into review items.

Disclosure norms help. Many teams add a note in the pull request when significant parts were assisted, not as a confession but so reviewers know where to look harder. Agree the norm together, and avoid using disclosure as blame.

WORKING QUESTION

Does your team have a shared set of request templates, or does every engineer reinvent the prompt?

RISKS AND CONTROLS

Security, privacy, intellectual property and review controls

05

The main risks are data leaving the building, insecure code entering the product, and reviewers relaxing because the code looks neat.

RISK	WHAT CAN GO WRONG	CONTROL
Secrets and data exposure	API keys, customer data or proprietary code pasted into an unapproved tool	Approved tools only; secret scanning; clear never-share list
Insecure code	Injection flaws, weak authentication, unsafe deserialization	Security review for sensitive areas; static analysis; reference OWASP guidance
Vulnerable or fake dependencies	Suggested packages that are outdated, malicious or nonexistent	Verify the package, publisher and version; dependency scanning
License and IP concerns	Output resembling third-party code, unclear terms of use	Follow legal guidance; check tool settings and your contracts; check current requirements
Prompt injection	Instructions hidden in files, issues on web pages that steer an agent	Limit agent permissions; treat external content as untrusted; human approval for actions
Excess permissions	Agents with broad repository, cloud or production access	Least privilege; sandbox; branch-only writes; no production credentials
Review fatigue	Large AI-generated diffs approved quickly	Smaller changes; reviewer checklists; sampling and audits

Security guidance such as the OWASP Top 10 and OWASP guidance on large language model applications is a sound reference for both the code you produce and the AI tools you adopt. Data protection obligations apply when personal data is involved, and contracts with customers may restrict where their data may be processed. Check current requirements and ask your security and legal colleagues instead of assuming.

Reviewer checklist for assisted changes

- Does the change do what the ticket asked and nothing more?
- Are all called functions, flags and packages real and used correctly in our versions?
- Are inputs validated and errors handled, including empty and hostile input?
- Do the tests fail when the behavior is broken, and not only pass when it is right?
- Is any secret, credential or personal data present in code, tests or logs?
- Were any checks, validations or permissions removed or loosened?
- Is the dependency list unchanged, or have new entries been vetted?
- Can the author explain the design in their own words?

MUTATION CHECK

To test AI-written tests, break the code on purpose, for example flip a condition, and confirm that the tests fail. Tests that never fail protect nothing.

WORKING QUESTION

What is the one category of data or code your team must never paste into an assistant, and is that written down where people will see it?



MEASURING VALUE

Measuring quality and value without fooling yourself

06

Speed is easy to measure and easy to misread. Pair it with quality and rework.

Engineers often feel faster with an assistant, and sometimes they are. Feelings are a poor measure, because time saved writing can be lost in debugging and review. A fair evaluation uses a mix of delivery measures, quality measures and developer experience, collected over several weeks, ideally comparing similar work with and without assistance.

MEASURE	WHAT IT SHOWS	CAUTION
Cycle time from start toEnd-to-end speed merge		Varies with task size; compare similar work
Review time and comment count	Burden on reviewers	Rising load can cancel author gains
Defect escape rate	Bugs found after release	Slow signal; track over quarters
Rework and reverts	Changes that needed redoing	Needs consistent tagging
Test quality, such as mutation score or failure on injected bugs	Whether tests are meaningful	Coverage percentage alone can mislead
Incident and security findings	Risk from assisted changes	Compare to a baseline, not to zero
Engineer survey	Satisfaction, trust and friction	Combine with objective data

01 Set a baseline
Record cycle time, review load and defect rates for a few weeks before wider adoption.

02 Tag assisted work
Use a light label on pull requests so you can compare outcomes honestly.

03 Compare like with like

Separate routine tasks from design-heavy ones. Assistance often helps the first group more.

04 Review monthly

Look at trends and examples with the team. Adjust standards, templates and tool access.



Illustrative example, not a benchmark: if a team's authors save an hour on a change but reviewers spend an extra 40 minutes untangling it, the net gain is small and the strain on reviewers is large. Look at the whole flow. Also watch for skill effects: junior engineers may ship more but learn less if they accept output without understanding it. Pair assisted work with explanation and mentoring.

AVOID VANITY METRICS

Lines of code produced, suggestions accepted and prompts per day measure activity. They say nothing about value and can reward the wrong behavior.



WORKING QUESTION

What would you measure to detect that AI is making your team faster at producing code that later costs more to maintain?



IMPLEMENTATION GUIDE

A rollout plan for an engineering team

07

Start with a small group, a written standard and low-risk use cases. Expand once the habits are visible in review.

01 Choose approved tools

With security and legal, pick tools and settings that meet your data rules. Decide who administers access.

02 Write the one-page standard

Cover allowed uses, never-share data, verification expectations, disclosure norm and where to report problems.

03 Pilot with volunteers

Pick six to ten engineers across experience levels for six weeks. Include a skeptic and a reviewer.

04 Build the template library

Collect prompts that worked, add the 'list assumptions' line, and store them in the repository.

05 Strengthen the safety net

Fix flaky tests, add static analysis, and make sure CI blocks risky changes. Assistance depends on this foundation.

06 Review and widen

Share examples of good and bad outcomes, update the standard, then extend to the next team.

Verification habits to teach

- Run before you read: execute the code and tests first, then review what passed and failed
- Check every external name: confirm packages, functions and flags in official documentation
- Read the diff end to end, including files you did not ask to change
- Try to break it: add a hostile input, an empty input and a boundary value

- Ask for the opposing view: have the tool argue why the change might be wrong, then evaluate the argument
- Keep the change small enough to review in one sitting

Team readiness

- Approved tools and access rules are in place
- The one-page standard is written and shared
- CI runs tests, linting and security scanning on every change
- Reviewers know to apply the same bar to assisted changes
- Secret scanning is active in repositories
- A baseline of delivery and quality measures exists
- A pilot owner and review date are set

For non-software engineering teams, the same shape applies: approved tools, a short standard, a pilot on document-heavy tasks such as test plans and review packs, and a firm rule that calculations and compliance conclusions are made with validated tools and checked by qualified engineers.

WORKING QUESTION

Who on your team will own the standard, and when will it be reviewed against real examples?

PITFALLS

Common pitfalls and an illustrative scenario

08

Most problems come from skipping review, widening permissions too fast and measuring the wrong thing.

- Accepting large diffs because they pass the tests, when the tests were also generated
- Letting agents run with broad credentials in shared environments
- Using AI for security-sensitive code without expert review
- Ignoring documentation drift because the draft looked finished
- Measuring suggestions accepted as success
- Allowing individuals to pick their own tools and rules with no team standard
- Neglecting onboarding, so juniors lean on output they cannot evaluate

ILLUSTRATIVE SCENARIO, HYPOTHETICAL

An engineer asks an assistant to add retry logic and tests. Both pass. In review, a colleague notices the retries wrap a non-idempotent payment call, so a timeout could charge twice. The fix is a design change, and the team adds idempotency to its reviewer checklist.

The scenario shows why tests that agree with the code are not proof. The tool did what was asked in the narrowest sense. The missing element was domain context: which operations are safe to repeat. This is the kind of constraint that belongs in the prompt, in the team standard and in the checklist, and it is why experienced reviewers remain essential.

SYMPTOM	LIKELY CAUSE	RESPONSE
Review queue grows	Larger, faster diffs	Cap change size; require author self-review notes
Bugs in assisted code	Missing verification habits	Teach run-first and break-it checks; add tests that fail

SYMPTOM	LIKELY CAUSE	RESPONSE
Security findings rise	Unreviewed sensitive code	Route sensitive areas to security review
Docs inaccurate	Drafts not checked against code	Require author to verify claims; add doc review
Uneven adoption	No shared templates or norms	Publish templates; share examples in team sessions

Cultural effects matter too. If speed becomes the only visible goal, engineers will stop admitting when they do not understand a generated change. Celebrate catches in review and treat honest uncertainty as a strength.

WORKING QUESTION

What is the riskiest place in your system where an assisted change could pass tests and still be wrong?

BUYING AND SETUP

Questions to ask when choosing and configuring tools

09

Choose on data handling, control and fit with your workflow, not on demos.

Questions for any vendor or internal platform team

- Is our code or data used to train models shared with others, and can we opt out contractually?
- Where is data processed and stored, and for how long are prompts and outputs retained?
- What admin controls exist for users, repositories, features and logging?
- Can we restrict the tool from specific repositories or file types, such as secrets and customer data?
- How does it handle agent permissions, command execution and network access?

- What is the support model and how are model changes communicated and tested?
- Does it integrate with our editor, version control, CI and ticketing?
- What are the total costs, including seats, usage, setup and evaluation time?
- What is the exit plan if we change tools?

CHOICE	ADVANTAGE	TRADE-OFF
Hosted assistant with enterprise terms	Fast start, managed updates	Depends on vendor data terms
Self-hosted or private deployment	More control of data	Operational cost and capability gap
Single tool for all	Simple to govern	May fit some teams less well
Several tools by team	Better fit per use	Harder governance and support

Run a structured trial on your own code and tasks. Define five or six representative tasks, run them across candidate tools, and score correctness, effort to verify, and fit with your conventions. Include a security-sensitive task and see how the tool handles it. Keep a record of results so the decision is explainable later.

If a second opinion would help, Halden & Company offers free assessments at halden.company/assessments, and you can contact the team at halden.company/contact. The principle to take away is unchanged: engineers remain accountable, verification is routine, and tools are chosen for control as much as capability.

WORKING QUESTION

If your assistant vendor changed its data terms next quarter, what would you need to know and decide within a week?

30 / 60 / 90

A first quarter that produces evidence

10

K'

Standardize the habits first, then expand access as review and measurement prove the quality holds.

First 30 days: ground it

- Select approved tools and write the one-page team standard
- Define never-share data and enable secret scanning
- Capture a baseline of cycle time, review load and defect rates
- Recruit a pilot group across experience levels



Days 31 to 60: test it

- Run the pilot on low-risk tasks such as tests, docs and refactors
- Build and share the request template library
- Add reviewer checklist items for assisted changes
- Hold a mid-pilot review of examples, good and bad



Days 61 to 90: decide

- Compare delivery and quality against the baseline
- Update the standard and widen to the next team
- Decide on agent permissions with security
- Schedule quarterly reviews of tools, terms and metrics



GLOSSARY

Terms used in this publication

11

TERM	MEANING
Prompt	The instruction and context given to an AI model.
Hallucination	Plausible output that is false, such as a function or reference that does not exist.
Agentic mode	A setting where the tool plans steps, edits files and runs commands with limited supervision.
Prompt injection	Hidden instructions in content that try to steer an AI tool into unwanted actions.
Mutation testing	Deliberately altering code to check whether tests detect the change.
Static analysis	Automated inspection of code for defects or vulnerabilities without running it.
OWASP	An open community that publishes widely used application security guidance.
Least privilege	Giving a person or tool only the access it needs for the task.
Idempotent	An operation that has the same effect whether performed once or several times.
Cycle time	The time from starting work to having it merged or delivered.
Secret scanning	Automated detection of credentials accidentally stored in code or text.
Definition of done	The agreed conditions a piece of work must meet to count as finished.

A note on use

Guidance is general and not legal, security or professional engineering certification advice. Qualified engineers remain responsible for calculations, safety cases and sign-off. Check current requirements and your contracts.

TAKE THE NEXT STEP

Take the free Halden assessment at halden.company/assessments, or talk to us at halden.company/contact about turning this into a plan for your organization.

