ChatGPT


```
{
  "summary": "Cursor has evolved into an AI-driven coding agent capable of planning tasks, generating code across files, running tests, and even fixing bugs autonomously. This research-based guide compiles best practices from mid-2025 on how to harness Cursor's full potential while minimizing errors and maintaining safety. Key strategies include using a **plan-then-execute** workflow for complex features [1] [2], scoping the AI's context with fine-grained file references and ignore rules [3] [4], adopting **TDD (Test-Driven Development)** so the agent writes and runs tests to verify its code [5] [6], and enforcing a strict **"safe edit" protocol** of minimal diffs, code reviews, and checkpoints [7] [8]. We also cover customizing Cursor's **rules engine** to encode project conventions, integrating external knowledge via **Docs/Web** retrieval, leveraging background agents for batch changes, and implementing security guardrails (secrets filtering, CWE checklists, sandboxed execution). These recommendations are grounded in real case studies – for example, Salesforce engineers boosted test coverage productivity by 85% using Cursor [6] – and quantitative trade-offs (like quality vs. cost vs. latency) observed in recent months. By following this playbook, teams can achieve high-reliability automation with Cursor while keeping human oversight and control in the loop.",
  "playbooks": {
    "plan_then_execute":
"## Plan-Then-Execute Strategy\n\n**Break tasks into a clear plan before coding.** For any sizeable feature or bug fix, have Cursor first outline the steps in natural language [1]. For example, ask *"What steps should we follow to implement X?"* and let it produce a checklist. This leverages the model's chain-of-thought to ensure it has a correct approach in mind [2]. Indeed, Cursor v1.2 introduced an automatic TODO list agent that internally plans tasks, but explicitly reviewing a plan with the human in the loop helps align expectations [1].\n\n**Execute in small, verified increments.** Rather than one big prompt for a complex change, guide Cursor through the plan step by step. This "divide and conquer" approach reduces errors and hallucinations [9]. For instance:\n1. *Plan:* "Add a login API endpoint" -> Steps: create route, add service logic, write tests.\n2. *Step 1:* *"Create the `/login` route in `auth.py`."* – Cursor adds the route code.\n3. *Step 2:* *"Implement JWT generation in the login service."* – Cursor writes logic.\n4. *Step 3:* *"Write tests for the login flow."* – Cursor generates tests and runs them.\n\nAfter each step, verify the output or run tests before moving on. Cursor will stay within the scope of the approved plan, reducing drift [10]. If something deviates (e.g. it wrote code unrelated to the current step), you can correct course immediately.\n\n**Leverage structured plans and rules.** You can also encode multi-step plans as Cursor **agents**. For example, one technique is writing a `.yaml` or `.mdc` file listing tasks (a "kanban board") and a rule for the agent to follow them [11]. Invoking `start @your-agent.mdc` triggers Cursor to sequentially execute the listed subtasks. This ensures the model doesn't skip critical steps. Always include clear success criteria for each step (e.g. "tests pass" or "UI matches design") so the
```

agent knows when to proceed.\n\n**Monitor and iterate.** Treat the AI's plan as a living spec – update it if requirements change or if you discover new substeps. Encourage the model to explain its reasoning if a step is unclear. If the agent stalls or seems confused mid-plan, you can prompt: *"Explain your plan so far and adjust if needed,"* then continue. Real-world use shows that investing a bit of time up-front in planning can save numerous back-and-forth edits [9] and prevent the AI from coding itself into a corner.",
    "context_scoping": "## Context Scoping and Referencing\n\n**Focus the assistant on relevant code.** Cursor indexes your codebase and can easily get overwhelmed if not directed. Use the **`@` references** to pinpoint context: e.g. mention `@utils/api.ts` instead of just saying \"the API code\" [12]. You can tag whole folders (like `@/models/` for a Django models directory) or specific files, which guides the AI to only fetch those parts of the project [13]. This dramatically improves accuracy in multi-file edits by narrowing the scope.\n\n**Use `.cursorignore` and `.cursorindexignore` to manage context.** The `.cursorignore` file excludes files from *all* AI visibility – use it for secrets, config files, or any area the AI shouldn't touch [14]. For example, add `.env`, `secrets/**`, build outputs, etc., to avoid accidental exposure of credentials [15] [14]. In contrast, `.cursorindexignore` stops files from being indexed into vector search but still allows you to explicitly refer to them via `@` [4]. A common pattern is to index source code but not large documentation files – you can keep docs out of the embedding index (to save context space) yet pull them in on demand with `@Docs` references.\n\n**Inject important background knowledge deliberately.** Instead of relying on the AI's training data (which may be outdated or generic), feed Cursor the specific knowledge it needs for your project. For example, if using a particular library, you can attach a summary of its usage in a Notepad and include it via `@Notepad` in prompts [16]. Cursor's **Memories** feature (beta in 1.0) can auto-save key facts from chat as long-term context [17] – review and approve these so the AI "remembers" project-specific details across sessions. Also consider writing a high-level architecture overview in the README or a rule file; users report that providing this upfront context helps the AI make more coherent decisions in large projects [18].\n\n**Retrieve external knowledge when needed.** Cursor can go beyond your repo using tools like `@Web` and `@Docs`. If the AI might need up-to-date information (say, an API reference or a StackOverflow solution), instruct it to use `@Web search X` – the assistant will fetch web results and incorporate them [19]. Or set up documentation MCPs (Model Context Plugins) such as *Context7* or official framework docs, so that `@Docs` brings in accurate, version-specific snippets [20]. This prevents hallucinations from stale training data by giving real documentation in-context. One team noted that using a docs MCP eliminated incorrect API guesses and reduced iteration time significantly [21] [22].\n\n**Multi-repo and monorepo strategies.** Cursor 0.50+ supports **multi-root workspaces**, meaning you can open multiple project folders in one Cursor session and it will index them all [23]. Take advantage of this for monorepos or projects split across services. Ensure each sub-project has its own `.cursor/rules` if conventions differ [24]. When prompting, be explicit about which project or module you're referring to ("In the *backend* service, …") so the AI loads the correct context. Cross-reference files between repos with `@` paths. By scoping queries to the right domain, you avoid confusion and get more relevant suggestions.\n\nIn summary, **be intentional with

context.** Guide Cursor to *where* in your codebase it should look, and *what* outside information to consider or ignore. This reduces noise and keeps its attention on the code that matters for the current task.",
    "tdd_as_spec": "## Test-Driven Development as Specification\n\nUsing tests as a spec greatly improves Cursor's reliability. By following a **Red → Green → Refactor** cycle, you force the AI to write code that meets explicit requirements:\n- **Red:** Start by writing a test (or instruct Cursor to generate a test) that defines the desired behavior. For a new feature, you might say, *"Write a failing unit test for function `foo` that exposes the bug/feature."* This ensures the AI fully understands the acceptance criteria. \n- **Green:** Next, have Cursor implement the minimal code to pass the test. Because the test is in context, Cursor will focus only on satisfying it. In Cursor, you can even turn on **Auto-Run (YOLO) mode** to let it run tests after each change [25]. For example, one developer prompted Cursor: "Run the build and fix any errors, repeating until it passes," and the agent automatically iterated – compiling, seeing errors, fixing code – until the build succeeded [26].\n- **Refactor:** Once tests pass, you can ask Cursor to clean up the code if needed (knowing the behavior is locked in by tests). Always re-run tests after refactoring to catch any regressions.\n\nThis approach gives much higher confidence in AI-generated code. Rather than accepting code that "looks right," you know it *works* because it's tested. A senior dev noted that unlike naive code suggestions, Cursor's code that passes a good test suite "comes with a guarantee" of correct behavior [5]. In practice, having the AI generate both tests and code can surface misunderstandings early – sometimes the first test it writes isn't quite what you intended, which is a cue to clarify the spec.\n\n**Enable automated test runs.** In Cursor's settings, enable Auto-Run and allow your test command (e.g. `pytest` or `npm test`) to run [27]. This way, after Cursor writes new code, it will execute the tests and see the failures without you copy-pasting output. Cursor reads the stack traces and immediately starts fixing the issues [27]. It will iterate until tests are green, or it runs into something it doesn't know how to fix. This tight loop dramatically speeds up debugging – essentially an AI pair-programmer running tests after each change. *Tip:* To avoid overwhelming the context window with long logs, run targeted subsets of tests when possible (e.g. a single test module) [28]. You can always run the full suite at the end.\n\n**Trust but verify the tests.** AI-generated tests can sometimes be superficial (e.g. asserting a function runs without error rather than checking correctness) [29] [30]. Review the tests critically – ensure they truly check the desired behavior and not just trivial conditions. You may need to guide Cursor: *"Add edge cases to the test"* or *"Assert the output value, not just that no exception is raised."* Once you have high-quality tests, let Cursor proceed to implementation. This was key in a case study where Salesforce used Cursor to raise legacy code coverage from <10% to 80% across 62 repos – engineers still oversaw the AI's test intentions and adjusted as needed for meaningful coverage [30] [6].\n\n**Benefits of TDD with Cursor:** Teams have reported massive productivity boosts using this method. The AI often finds and fixes issues that would be hard to spot otherwise. For example, an AI-generated test at Salesforce exposed a logic bug in production code (an AND vs OR in a health-check) that developers hadn't noticed [31]. By writing tests first, the AI not only met the spec faster but also acted as a code reviewer, flagging hidden bugs. Another benefit is consistency – once

you perfect a test for one module, you can tell Cursor to apply the same pattern to dozens of similar modules, and it will reliably reproduce the correct code (one team generated 180k lines of unit tests in 12 days by templating and cloning test patterns [32] [33]).\n\nIn summary, treat tests as the contract and let the AI fulfill that contract. This aligns the AI's objective with yours (passing tests), avoids ambiguity, and results in much higher quality code. When Cursor outputs "All tests passed," you can be much more confident in merging those changes.",

"safe_edit_protocol": "## Safe Edit Protocol and Quality Control\n\nWhen letting Cursor modify code, **keep a tight leash on changes** to prevent "AI drift" or unintended side effects:\n- **Work in small diffs.** Aim to have Cursor change only what's necessary for the task at hand. You can achieve this by scoping prompts narrowly (e.g. *"Update the `addUser` function to hash the password, and *only* that function"*) and by explicitly telling the AI not to stray: e.g. *"Do not alter anything else."* [8] . Cursor will then confine its edits to the relevant section. If the suggestion still comes back too large or modifies unrelated files, don't accept it – refine the prompt or break the task down further. It's often wise to tackle one file at a time; the Salesforce team learned to abandon broad instructions that changed hundreds of files at once (which forced a full rollback) and instead guide Cursor class-by-class [29] .\n- **Use diffs and incremental review.** Cursor presents changes as diffs which you can selectively accept or reject. Always inspect these diffs. Ensure the changes make sense and fit coding standards before hitting "Apply" [7] . It helps to have tests (as mentioned) to validate the changes. If your toolchain supports it, run linters and formatters on the diff as well – or ask Cursor to fix any lint issues before accepting. Cursor's *Restore Checkpoint* feature lets you revert an applied change if you realize it was problematic [34] , but it can be glitchy, so it's best to catch issues **before** applying. A good habit is to commit your code (or use a Git branch) prior to large AI edits, so you have an easy manual fallback via `git reset` [35] .\n- **One change per review.** Resist the temptation to let Cursor stack multiple logical changes in one go. For example, don't combine a major refactor with a new feature in a single prompt. Do them separately, testing in between. This protocol of incremental, isolated changes makes it far easier to pinpoint and fix any bug introduced by the AI. It also aligns with human code review best practices – small PRs are easier to understand and validate. Cursor's background agents follow this by opening a fresh branch/PR for each task [36] , so you can code-review the AI's work just like you would a teammate's.\n- **Bring in automated reviewers.** Treat AI-generated code with the same skepticism as a human's code. Enable CI checks on AI-authored branches and consider using Cursor's **BugBot** or similar tools to audit the diff [37] . BugBot will comment on a PR with potential issues or confirm the changes look sound. While it's an AI itself, it's trained for code review and can catch things the code-writing model might miss (like runtime complexities or corner-case bugs). In one scenario, Cursor suggested introducing a new dependency – the dev team paused and checked the package's reputation, discovering it was a typosquat (malicious fork) [38] . The lesson: always vet AI-suggested dependencies and changes with security in mind.\n- **Establish guardrails in rules.** You can write Cursor rules that enforce safety checks – for example, a rule that forbids modification of critical config files, or one that warns "don't call `exec()` or install packages" in

```
most contexts. While the AI might occasionally override or ignore rules if not strictly enforced, these rules provide persistent reminders in the model's context [39] [40]. Also, if you have sensitive sections of code (payment processing, credential management), consider adding comments like "// @cursor: do not modify without approval" – these can act as soft guardrails. \n- **Sandbox testing for risky changes.** For any change that touches security, infrastructure, or other high-risk areas, use a sandbox. This could mean running the AI's changes on a staging environment first, or at least isolating the changes on a branch and reviewing them with senior engineers. It's recommended to **disable YOLO mode in critical repositories** [41] so that the AI can't execute potentially dangerous commands or deploy code without human oversight. In enterprise settings, restrict Cursor's permissions: e.g. it should not have credentials to production systems, and its PRs should require human approval. Treat the AI like a junior developer with root access – powerful but in need of supervision [42].\n\nBy adhering to this "safe edit" protocol, you ensure that using Cursor accelerates development **without** introducing unacceptable risk. Every AI-generated change is reviewed (by you or another tool), tested, and easy to revert if something goes wrong. In practice, this dramatically reduces the chances of an AI mistake making it into your main codebase. For instance, Salesforce enforced strict source code boundaries and a SonarQube gating check on Cursor's contributions, successfully avoiding regressions while integrating tens of thousands of AI-written lines [43]. With the right safeguards, you can embrace Cursor's speed and still sleep soundly, knowing nothing gets in without a second set of (human or automated) eyes."
  },
  "rules": [
    {
      "stack": "react-ts",
      "filename": "react-guidelines.mdc",
      "content": "---\ndescription: \"Enforce React/TypeScript coding conventions\"\nglobs: \"src/**/*.tsx\"\nalwaysApply: false\n---\n\n# React & TypeScript Conventions\n\n- Use **functional components** and Hooks (no class components) for all new React code.\n- Organize components by feature: e.g. place each component in `src/components/<feature>/` with related files [44].\n- Use **TypeScript** for type safety. Define explicit interfaces or types for component props; avoid using `any`.\n- **Tailwind CSS** is the primary styling method – prefer utility classes over custom CSS, and avoid inline styles (keep UI consistent) [44].\n- Follow our linting/formatting rules (ESLint, Prettier). Code should have no linter errors or warnings.\n- Use **camelCase** for variables and functions [45] and **PascalCase** for React component names.\n- Prefer `axios` over `fetch` for HTTP requests (for consistency in error handling).\n- Do not use deprecated React APIs (no old lifecycle methods like `componentWillMount`); use Hooks or modern equivalents.\n- Ensure that all component state updates are done via Hooks (`useState`, `useReducer`, etc.) rather than legacy techniques.\n- Include basic **unit tests** for each component's critical logic (e.g. using React Testing Library). Tests help verify that UI and props behave as expected.\n"
    },
    {
      "stack": "fastapi",
```

```
      "filename": "fastapi-guidelines.mdc",
      "content": "---\ndescription: \"Python FastAPI coding
standards\"\nglobs: \"**/*.py\"\nalwaysApply: false\n---\n\n# FastAPI
Guidelines\n\n- **PEP 8** – Adhere to Python coding style (snake_case names,
4-space indents, etc.). Use type hints on all functions, especially for
request/response schemas.\n- Structure the API with **APIRouter** modules.
Don't put all endpoints in `main.py` – split by resource (e.g. `users.py`,
`orders.py`).\n- Use **Pydantic models** for request bodies and responses.
Validate all incoming data with these models (FastAPI will auto-generate docs
from them).\n- Utilize FastAPI's **dependency injection** with `Depends` for
shared logic (database sessions, auth). Avoid global variables for request-
specific context.\n- Each endpoint should handle errors gracefully: use
`HTTPException` for client errors (400/404/etc.) and let FastAPI handle
server errors (500) with proper logging.\n- **Async** endpoints: If an
endpoint performs I/O (database calls, HTTP requests), define it with `async
def` and use awaitable libraries. This keeps throughput high.\n- No sensitive
secrets (DB passwords, API keys) hard-coded in settings or code. Retrieve
them from environment variables or a secrets manager. Mark any secret-config
files in `.cursorignore`.\n- Add **docstrings** to endpoints explaining their
behavior (these can become API docs). Document especially any security
aspects (auth required, scopes).\n- Write **unit tests** for critical logic
(e.g. authentication, business rules). Use pytest with FastAPI's test client
to simulate requests and ensure endpoints work as expected.\n"
    },
    {
      "stack": "node-express",
      "filename": "express-guidelines.mdc",
      "content": "---\ndescription: \"Node.js/Express coding
guidelines\"\nglobs: \"**/*.{js,ts}\"\nalwaysApply: false\n---\n\n# Express
Guidelines\n\n- Use modern JavaScript/TypeScript features. Prefer `const`/
`let` and arrow functions; avoid legacy ES5 syntax.\n- **Modularize routes**
using Express Router. Group related endpoints in their own router (e.g.
`routes/user.js`) and use `app.use('/users', userRouter)`.\n- Validate client
input on every route. Use middleware (like celebrate/Joi or express-
validator) to enforce schema and prevent invalid data (helps avoid
injections).\n- Implement centralized error handling. Use an Express error-
handling middleware to catch exceptions from routes (don't let crashes
propagate).\n- No blocking operations in request handlers. Intensive CPU
tasks should be offloaded (e.g. to a worker thread or queue) so they don't
block the event loop.\n- Do not expose sensitive information. Strip out stack
traces or secrets from error responses (send generic error messages to
clients).\n- **Security headers:** Use `helmet()` middleware to set secure
HTTP headers. Enable CORS only for allowed origins.\n- Load configuration
from environment variables (using a library like dotenv for local dev). Never
commit secrets or credentials to the repo.\n- Use `async/await` (or promises)
for all async code – avoid callback patterns. Handle promise rejections to
prevent unhandled rejections.\n- Write basic integration tests for routes
(e.g. using Supertest) to ensure endpoints respond as expected (200 on
success, proper error codes on failure).\n"
    },
    {
```

```
      "stack": "other",
      "filename": "secure-coding.mdc",
      "content": "---\ndescription: \"General secure coding best
practices\"\nglobs: \"*\"\nalwaysApply: true\n---\n\n# Secure Coding
Guidelines (All Projects)\n\n- **Input Validation:** Treat all external input
as untrusted. Validate inputs (HTTP requests, form fields, API payloads) for
type, format, length, etc., before using them. Reject or sanitize anything
unexpected.\n- **Output Encoding:** Escape or encode outputs to prevent
injection attacks. For example, escape HTML in web responses to prevent XSS
(Cross-Site Scripting) and use parameterized queries or ORMs to prevent SQL
injection.\n- **Error Handling:** Handle errors gracefully and securely. Do
not expose internal error details or stack traces to users. Log errors on the
server, but return generic error messages to avoid leaking information.\n-
**Authentication & Authorization:** Enforce authentication on all appropriate
endpoints. Check user roles/permissions (authorization) before allowing
actions. Never assume security on the client side – always re-verify on the
server.\n- **Cryptography:** Use established libraries for crypto
(encryption, hashing). Never hard-code cryptographic keys or secrets in code.
Rotate secrets periodically and store them securely (e.g. in environment
variables or vault services).\n- **Session Security:** If using sessions or
JWTs, ensure they are secure (HttpOnly cookies, proper expiration). Validate
JWT signatures and claims. Implement rate limiting or lockouts on login to
mitigate brute-force attacks.\n- **File I/O Safety:** When dealing with file
uploads or file system access, use safe file handling. Store files in
designated directories, and prevent path traversal (don't accept "../" in
filenames, etc.).\n- **Dependency Vigilance:** Keep dependencies up to date.
Avoid packages with known vulnerabilities (use tools like `npm audit` or `pip
audit`). Be cautious of installing unknown or unmaintained libraries – they
could be malicious [38].\n- **Secure Defaults:** Follow security best practices
by default – e.g., secure passwords (hash with a strong algorithm like
bcrypt), enable TLS/HTTPS, and set secure cookie flags. \n- **Monitoring:**
Log security-relevant events (logins, permission denials, etc.) and monitor
logs for anomalies. Employ automated security tests (like SAST/DAST tools) to
catch common vulnerabilities (CWE patterns) before shipping.\n"
    }
  ],
  "prompts": [
    {
      "name": "ConvertToFunctional",
      "mode": "inline",
      "content": "Convert this React class component to a functional
component using Hooks, preserving its existing behavior and props."
    },
    {
      "name": "AddAuthEndpoint",
      "mode": "composer",
      "content": "Implement a new **`/login`** API endpoint in the FastAPI
app. It should accept a username and password, verify the credentials against
our `User` database, and return a JWT on success. Use Pydantic models for the
request and response schemas. Please provide any new code (routes, schemas,
auth logic) required for this feature."
```

```
    },
    {
      "name": "EnableCaching",
      "mode": "composer",
      "content": "Add caching to the GET `/items` endpoint in our Express
app. Use an in-memory cache so that if the same request comes again, we
return cached data instead of hitting the database. Cache entries should
expire after 5 minutes. Outline the necessary changes and implement them."
    },
    {
      "name": "SecurityReview",
      "mode": "composer",
      "content": "Review the above code for security vulnerabilities and bad
practices. List any issues you find (e.g. SQL injection, XSS, leaked secrets,
etc.) and suggest how to fix them."
    }
  ],
  "model_matrix": [
    {
      "task": "Inline code completion or quick refactor",
      "model": "Cursor Fireworks (fine-tuned code model)",
      "params": {
        "temperature": 0.1,
        "max_tokens": 100
      },
      "expected_cost_latency": "Negligible cost (included in subscription);
very low latency (typically <1s per suggestion) [46] . Optimized for single-file
context."
    },
    {
      "task": "Chat assistant (moderate multi-file task)",
      "model": "OpenAI GPT-3.5 Turbo (16k context)",
      "params": {
        "temperature": 0.2,
        "max_tokens": 1000
      },
      "expected_cost_latency": "Approx. $0.002 per 1K tokens (for reference)
- extremely cheap [47] . Usually responds in a few seconds for medium tasks.
Good for everyday coding queries with moderate reasoning."
    },
    {
      "task": "Complex refactor or large feature (agent mode)",
      "model": "OpenAI GPT-4 (32k) or Claude 2 (100k) in Cursor Max mode",
      "params": {
        "temperature": 0.0,
        "max_tokens": 20000
      },
      "expected_cost_latency": "High cost (~50× higher than GPT-3.5; e.g.
$0.06–$0.12 per 1K tokens) and higher latency (15–30s or more) [48] . Used when
extensive context or superior reasoning is needed. Cursor may limit context
used (~70–120k vs 200k max) to keep latency reasonable [49] ."
```

```
    },
    {
      "task": "Codebase indexing and semantic search",
      "model": "OpenAI Embeddings (Ada-002)",
      "params": {
        "temperature": 0.0,
        "max_tokens": 2048
      },
      "expected_cost_latency": "Very low cost (~$0.0004 per 1K tokens for
embeddings). Indexing an entire repository (e.g. 100K tokens) costs only a
few cents and takes a couple of minutes in the background. Enables instant
semantic code search with minimal overhead."
    }
  ],
  "kpis": {
    "definitions": [

"Green Test Percentage – The fraction of AI changes that pass all tests
(ideally on the first run). Higher is better (indicates the AI produced
correct code without needing fixes).",
      "Time-to-Green – How long it takes for an AI task to go from start to
all tests passing. Measured per task (in minutes or hours). This reflects
velocity; shorter times mean faster delivery.",
      "Average Diff Size – The average size of AI-generated code changes
(lines added/removed). Smaller diffs reduce review effort and risk. We track
this to ensure the AI isn't making excessive or unrelated edits.",
      "Iterations per Task – The average number of edit cycles or prompts
needed for the AI to complete a task. Fewer iterations mean the AI was more
efficient and needed less back-and-forth.",

"Cost per 100 Lines – The approximate API cost to generate 100 lines of code.
This normalizes expense against output. (For example, if 100 lines costs
$0.05 in API calls, that's the metric.)"
    ],
    "collection": "We collect these KPIs using automated hooks in our
development workflow:\n- **Green Test % and Time-to-Green:** Every time
Cursor opens a pull request or completes a task, our CI runs the test suite.
We log whether it passed outright or how many runs it took. The duration from
task start to the first green test run is recorded as Time-to-Green. Over
many tasks, we calculate the percentage that passed all tests on first
attempt. (In one case study, adopting AI improved dev time per module from 26
days to 4 days on average – an 85% reduction [6], and later tasks achieved
~90% first-try success by reusing proven patterns [32].)\n- **Diff Size and
Iterations:** We use git metadata and Cursor's conversation logs. After each
AI commit, a script parses the `git diff --stat` to count lines changed. It
also counts the number of messages/edits the AI made (each "Assistant
proposes a change" counts as one iteration). We monitor that diffs stay small
(if average lines changed starts creeping up, it might indicate the AI is
doing too much in one go or including unrelated edits). Likewise, a spike in
iterations per task might signal the model is struggling (possibly due to
context issues or too high complexity in one session).\n- **Cost per 100
```

```
Lines:** We integrate with Cursor's usage metrics (the token counts from the
OpenAI/Anthropic API calls). For each task, we compute cost = tokens_used *
price_per_token. We also know the output size (lines of code). Over a span,
we aggregate total cost and total lines produced to get an average cost per
100 lines. This helps management budget AI usage. In our experience, this has
been very economical – e.g. one developer had Cursor generate ~210k lines for
about $40 [50], which is ~$0.02 per 100 lines, a huge value compared to manual
coding.\n"
  },
  "security": {
    "secrets_handling":
"All secrets and credentials are strictly kept out of AI context. We maintain
a robust **.cursorignore** that includes any secret files or keys (API
tokens, `.env` files, private keys, etc.) [14]. This ensures the AI cannot
accidentally read or leak them. We also make sure not to open secret files in
the editor while Cursor is enabled [15] (Cursor can read any open file). If we
need to discuss sensitive info (e.g. error logs containing user data) with
the AI, we **redact or replace** the sensitive parts beforehand [51]. Cursor's
own safety mechanisms will try to detect and mask obvious secrets (and its
indexing avoids certain patterns by default) [52], but we don't rely solely on
that – we proactively sanitize context. Additionally, secrets are stored in
environment variables or vaults; the AI is instructed to use config
placeholders and never hardcode secrets in code.",
    "cwe_checklist": [

"CWE-79 (Cross-Site Scripting): Always escape or sanitize user-provided
content before rendering in a page or email to prevent XSS.",
      "CWE-89 (SQL Injection): Use parameterized queries or ORM methods for
all database access. Never directly concatenate untrusted input into SQL
statements.",

"CWE-78 (OS Command Injection): Avoid passing user input to system/exec
commands. If needed, whitelist allowable inputs and use safe libraries for
shell interactions.",
      "CWE-352 (Cross-Site Request Forgery): Protect state-changing actions
with CSRF tokens or same-site cookies. Verify the origin of important
requests.",
      "CWE-200 (Information Exposure): Do not leak sensitive data in logs or
error messages. Mask personal info and credentials. Use HTTPS to protect data
in transit.",
      "CWE-798 (Use of Hard-coded Credentials): Never embed passwords, API
keys, or secrets in code. Use environment configs and secure secret storage
instead."
    ],
    "pot_sandbox": "We run AI-driven operations in isolated environments to
mitigate risk. Cursor's **Background Agents** already execute on sandboxed
cloud VMs (separate from the developer's machine) [53], which limits any
potential damage from running code. For local development, we use
containerized dev environments and restrict the AI's access (for example, no
prod database credentials are present in the dev environment). We've also
disabled fully autonomous actions on critical infrastructure – the AI can
```

```
propose changes or run tests, but deployments require human approval [41]. Essentially, we treat the AI like untrusted code: we confine its actions to a safe sandbox and review anything it produces before it goes to production. This includes using source control protections (AI branches cannot directly merge to main without code review) and enabling Workspace Trust settings where possible to prevent unauthorized tasks. By sandboxing Cursor's activities, we ensure that even if it tries something unintended, it cannot affect sensitive systems or data."
  },
  "failures": [
    {
      "name": "Context Drift in Long Sessions",
      "symptoms": [
        "AI responses become irrelevant or contradictory in a long chat session.",
        "Cursor forgets earlier instructions or code context (e.g. reintroduces a bug you already fixed).",
        "Model starts making broad suggestions that don't align with the task at hand."
      ],
      "fix_prompts": [
        "Start a **new chat session** for the task. Long sessions can degrade quality; Cursor itself warns that after many turns, context can be lost [54]. Re-summarize the important details in the new session.",
        "Use **smaller, focused prompts**. If the conversation was meandering, break the problem into sub-tasks and tackle them one by one, each in its own fresh prompt. This keeps context concise.",
        "Explicitly **remind** the AI of key context in the new chat (e.g. reattach important files with @ references or restate the goal and constraints)."
      ]
    },
    {
      "name": "Unintended or Broad File Changes",
      "symptoms": [
        "Cursor modifies files or code that you didn't ask it to change.",
        "The AI's diff includes unrelated refactors or updates (e.g. changing formatting in untouched files, updating package versions without being asked).",
        "It "runs away" with a task – applying a change globally when you intended it to be local."
      ],
      "fix_prompts": [
        "Tighten the **scope** in your prompt. Add a line like "Only change the code in `<specific file/function>` and nothing else." This often prevents scope creep [8].",
        "Use **rules or ignore files** to fence it in – e.g. add non-target files to `.cursorignore` temporarily, so it literally can't see or edit
```

```
them.",
        "Review the plan it's following: you might say, "List the files you
plan to modify." If it lists more than expected, correct it before
execution."
      ]
    },
    {
      "name": "Stuck in a Fix Loop",
      "symptoms": [
        "Cursor keeps making changes and running tests in a loop, but errors
persist or toggle between states.",
        "The AI fixes one thing, but that causes a new failure, then fixes
that, causing another – it's not converging to a solution.",
        "Progress has stalled – the same few tests fail repeatedly despite
multiple fixes."
      ],
      "fix_prompts": [
        "Break the cycle by giving a **strategic hint**. For example: "The
error is coming from the date parsing logic – focus on that." A nudge can get
the AI out of a rut.",
        "Simplify the scenario. Stop the auto-run and ask Cursor to explain
the problem in its own words. Sometimes having it summarize the issue reveals
a misunderstanding that you can then clarify.",
        "If all else fails, **reset the conversation** and approach the
problem differently. You might tackle a smaller portion of the issue first.
(The Salesforce team found they had to intervene and reset when Cursor got
stuck in compile-fix loops [55] .)"
      ]
    },
    {
      "name": "Model Cannot Resolve Issue (Blind Spot)",
      "symptoms": [
        "The AI repeatedly gives an incorrect solution or says it cannot fix
the issue.",
        "Cursor's suggestions circle around an obvious bug without fixing it
(or it insists there is no bug when there is).",
        "It might produce the same code again or simply stall with an
apology."
      ],
      "fix_prompts": [
        "Try a **different model** for this problem. What GPT-4 doesn't
catch, Claude might – and vice versa [56] . Different LLMs have different
training data and strengths.",
        "Provide the exact **error message or failing scenario** again and
explicitly say "Fix this." [57] Sometimes a direct approach with the raw error
helps the AI zero in on the solution.",
        "Double-check if the issue is due to missing context. If so, feed in
the relevant code or info it might be lacking. You can also ask the AI to
explain the code back to you – its explanation might reveal a misconception
that you can correct in your prompt."
      ]
```

```
    }
  ],
  "references": [
    {
      "id": "Kara2025",
      "title": "Cursor AI Complete Guide (2025): Real Experiences, Pro Tips,
MCPs, Rules & Context Engineering",
      "url": "https://medium.com/@hilalkara.dev/cursor-ai-complete-
guide-2025-real-experiences-pro-tips-mcps-rules-context-
engineering-6de1a776a8af",
      "type": "post",
      "year": 2025
    },
    {
      "id": "Singh2025",
      "title": "How Cursor AI Cut Legacy Code Coverage Time by 85%",
      "url": "https://engineering.salesforce.com/how-cursor-ai-cut-legacy-
code-coverage-time-by-85/",
      "type": "post",
      "year": 2025
    },
    {
      "id": "ByteByteGo2025",
      "title": "How Cursor Serves Billions of AI Code Completions Every Day",
      "url": "https://blog.bytebytego.com/p/how-cursor-serves-billions-of-
ai",
      "type": "post",
      "year": 2025
    },
    {
      "id": "Kern2025",
      "title": "14 Practical Cursor Tips From Daily Use",
      "url": "https://www.instructa.ai/blog/cursor-ai/cursor-pro-tips-2025",
      "type": "post",
      "year": 2025
    },
    {
      "id": "Munn2025",
      "title": "Cursor AI Security - Deep Dive into Risk, Policy, and
Practice",
      "url": "https://dev.to/tawe/cursor-ai-security-deep-dive-into-risk-
policy-and-practice-4epp",
      "type": "post",
      "year": 2025
    },
    {
      "id": "Qodo2025",
      "title": "Claude Code vs Cursor: Deep Comparison for Dev Teams (2025)",
      "url": "https://www.qodo.ai/blog/claude-code-vs-cursor/",
      "type": "post",
      "year": 2025
```

```
    },
    {
      "id": "Sidetool2025",
      "title": "Cursor AI Cost Analysis: Evaluating the Value for
Developers",
      "url": "https://www.sidetool.co/post/cursor-ai-cost-analysis-
evaluating-the-value-for-developers/",
      "type": "post",
      "year": 2025
    },
    {
      "id": "Sewell2025",
      "title": "How I use Cursor (+ my best tips)",
      "url": "https://www.builder.io/blog/cursor-tips",
      "type": "post",
      "year": 2025
    }
  ],
  "meta": {
    "time_window": "2025-06-01 to 2025-09-30",
    "source_mix": {
      "docs": 1,
      "repos": 0,
      "case_studies": 1,
      "benchmarks": 1
    },
    "confidence_notes": "Our findings draw on a range of recent (mid-2025)
sources including official guides, engineering case studies, and community
best-practice posts. There is **strong consensus** on the core workflows
(planning, context management, TDD, etc.), which are corroborated by multiple
independent sources (e.g. both a Salesforce engineering post and a Medium
guide emphasize iterative planning and small diffs). Where direct
documentation was unavailable (e.g. exact internal model parameters), we
inferred details based on known usage patterns and OpenAI/Anthropic pricing
as of 2025. We are confident in the recommendations as they are backed by
real-world case studies (like an 85% productivity boost reported 6 ) and
explicit statements from Cursor's own community experts. One area of
interpretation was **sandboxing ("PoT sandbox")** – since no source used that
exact term, we aligned our answer with sandboxing best practices mentioned
across security discussions 41 53 . Overall, each key point (from rules
writing to model selection trade-offs) is supported by evidence, and any
extrapolation (such as specific KPI formulas) is noted as a synthesis of best
practices rather than verbatim from a single source."
  }
}
```

1 2 3 8 9 10 12 16 21 22 36 37 39 40 44 45 54 56 57 Cursor AI Guide (2025): MCPs, Rules, Tips & Real Results | Hilal Kara | Medium
https://medium.com/@hilalkara.dev/cursor-ai-complete-guide-2025-real-experiences-pro-tips-mcps-rules-context-engineering-6de1a776a8af

4 7 11 13 20 34 35 14 Practical Cursor Tips From Daily Use | Instructa Courses
https://www.instructa.ai/blog/cursor-ai/cursor-pro-tips-2025

5 25 26 How I use Cursor (+ my best tips)
https://www.builder.io/blog/cursor-tips

6 29 30 31 32 33 43 55 How Cursor AI Cut Legacy Code Coverage Time by 85%
https://engineering.salesforce.com/how-cursor-ai-cut-legacy-code-coverage-time-by-85/

14 15 38 41 42 51 Cursor AI Security - Deep Dive into Risk, Policy, and Practice - DEV Community
https://dev.to/tawe/cursor-ai-security-deep-dive-into-risk-policy-and-practice-4epp

17 19 46 52 53 How Cursor Serves Billions of AI Code Completions Every Day
https://blog.bytebytego.com/p/how-cursor-serves-billions-of-ai

18 23 24 Manage Repos & Large Codebases in Cursor | Instructa Courses
https://www.instructa.ai/blog/cursor-ai/how-to-multiple-repository-and-large-codebase-in-cursor

27 28 Cursor terminal access for auto-fixing unit tests - How To - Cursor - Community Forum
https://forum.cursor.com/t/cursor-terminal-access-for-auto-fixing-unit-tests/69922

47 50 Cursor AI Cost Analysis: Evaluating the Value for Developers
https://www.sidetool.co/post/cursor-ai-cost-analysis-evaluating-the-value-for-developers/

48 49 Claude Code vs Cursor: Deep Comparison for Dev Teams [2025] - Qodo
https://www.qodo.ai/blog/claude-code-vs-cursor/