ChatGPT

```
{
  "summary": "A **Cursor coding agent** can reliably implement code changes
by planning tasks, executing minimal diffs, and self-testing in cycles [1] [2] .
We feed the agent persistent context like **Project Rules** and docs, so it
understands our architecture and conventions [3] [4] . The agent works through a
**Plan→Code→Test→Review loop**: first breaking a feature into a checklist of
steps [1] , then coding each step, running tests or linters, and reviewing
results. This structured autonomy prevents drift and keeps changes scoped and
safe. We also integrate **tools** (linters, test runners, etc.) in the loop
so the agent catches errors early and fixes them before finalizing changes [5] .
Real-world cases show massive productivity gains (e.g. 85% reduction in test
writing time) when using these methods [6] . The following deliverables provide
playbooks, rules, prompts, and configurations to deploy a production-grade
Cursor Agent, along with measurable KPIs (e.g. test pass rate, iteration
count, cost) to track its performance.",
  "playbooks": {
    "plan_then_execute": "**Plan Then Execute:** In Agent mode, Cursor can
**generate a step-by-step plan** for complex tasks before writing any code [1] .
This plan is presented as a checklist of actions (e.g. create a module,
update a config, write tests). **Review or refine the plan** with the agent
to ensure all requirements are covered. Once confirmed, the agent executes
each step in sequence, checking off tasks as they complete. This approach
uses an autonomous loop that **reads the plan, acts, and verifies** in each
iteration [7] . It dramatically reduces errors by ensuring the AI “**thinks
step by step**” and avoids rushing into code [1] . If the agent deviates or a
step fails, it can revisit or adjust the plan rather than continuing blindly.
In practice, this yielded better outcomes than one-shot prompts, as observed
by users who note that planning first leads to *\"much more structured
output\"* [1] . *Templates:* We include a `/plan` command prompt that tells the
agent to draft a numbered to-do list for any high-level request, which you or
the agent can then approve and execute.",
    "context_scoping": "**Context Scoping:** To keep the AI focused and
within context limits, provide only the **relevant code and docs** for each
task. Cursor’s design loads files on demand, so open the files or folders
that the agent needs to see. For large projects, manually opening key files
ensures they’re within the ~8k token context; otherwise the agent might miss
important parts of a monorepo [8] . Use specific @file references in prompts
instead of broad project scopes, as **targeted context yields better
results** [9] . A `.cursorignore` file helps exclude irrelevant or sensitive
files from the AI’s view [10] . For instance, add environment configs, large
data files, or auto-generated code to **keep them out of context** [10] .
Leverage **@Docs or @Web** to inject just-in-time documentation (via tools
like Context7 MCP) for any library or API the agent uses, so it doesn’t
hallucinate APIs [11] . In summary, **feed minimal, high-signal context**: the
files to change, any dependent interfaces, and concise project rules. This
keeps the model’s attention on what matters and prevents confusion.
*Pitfall:* On multi-million line codebases, purely agentic search can slow
```

```
down (e.g. 2–5 minutes) and incur cost [12] . In such cases, pre-indexing
(Vector databases or Cody-like embeddings) or splitting work per module might
be necessary, though Cursor's built-in search is usually sufficient up to
medium-size repos.",
    "tdd_as_spec": "**TDD as Spec:** Treat test cases as the specification
for the code. **Have the agent write tests first**, then code to make them
pass [13] . This keeps it *"on rails"* and prevents feature creep. For a new
feature or bug fix, start by prompting the agent (in Composer) to generate a
failing unit test that describes the expected behavior or reproduces the bug.
Once the test is in place, instruct the agent to implement the code until the
test passes (the Red→Green cycle). Cursor can run tests in the integrated
terminal (especially with Auto-Run enabled), giving immediate feedback [5] . By
iterating this way, the agent knows when it's done (all tests green). This
workflow also yields better designs: in one case, Cursor's AI-generated tests
even uncovered a logic bug in existing code (AND vs OR mistake) that devs
hadn't noticed [14] . **Scope limits:** keep each test small and focused on one
behavior; if the agent tries to solve too much at once, break it into
additional tests. We provide a `/tdd` command that automates this pattern: it
creates a skeleton test file from a feature description, then prompts the
agent to fill in implementation and run the tests. *Pitfall:* Ensure the
agent doesn't delete or bypass tests. Always run the full test suite after
agent changes to catch any regressions.",
    "safe_edit_protocol": "**Safe Edit Protocol:** The agent should make
**minimal, reviewable diffs** and use confirmation checkpoints. Enable **diff
mode** so you see exactly what changes are proposed. Our rules enforce that
the AI should *\"only modify the intended sections\"* and not rewrite
unrelated code. After each edit, the agent (or CI hook) runs linters and
tests. If something fails, the agent fixes it before proceeding. This tight
feedback loop catches errors early [5] . Use Cursor's **Auto-Run (YOLO) mode**
carefully – allow it to run tests or formatters automatically, but **restrict
dangerous commands** (e.g., no `rm -rf` or external network calls) via the
allowlist in settings [5] . The 1.7 update introduced **Hooks** that let us
intercept agent actions: e.g., we can block any file deletes or secret prints
at runtime [15] . We include a sample hook script to require manual approval if
the agent tries to modify dependency files or config. Finally, always have
version control as a safety net: the agent should commit after each major
step (we have a `/pr` command that creates a PR with a summary of changes).
In case of mishaps, you can revert. A Salesforce team noted that initially
their agent *"went rogue, modifying hundreds of files…requiring complete
rollbacks"*, so they revised their approach to focus on one module at a time
 [2] . This protocol institutionalizes that lesson: **small, incremental edits,
with human checkpoints**, to maintain control and code integrity."
  },
  "rules": [
    {
      "stack": "react-ts",
      "filename": "frontend-conventions.mdc",
      "content":
"--\ndescription: React/TypeScript UI coding conventions and best
practices\nglobs:\n  - \"src/frontend/**\"\n  - \"**/*.tsx\"\nalwaysApply:
false\n--\n**Project Structure & Exports**\n- Place each React component in
```

```
its own file; use **named exports** (no default exports).\n- Organize
components by feature in `src/frontend/components/[feature]/` directories.
\n\n**Styling**\n- Use Tailwind CSS utility classes for styling, but prefer
semantic class names (e.g. `btn-primary`).\n- Avoid inline styles. Use shared
style modules for common patterns.\n\n**Coding Style**\n- Use **functional
components** and React Hooks; avoid class components [16] .\n- All components
and hooks should be written in **TypeScript** with explicit types for props
and state.\n- Name state variables with intent (e.g. `isLoading`, `hasError`)
[17] .\n\n**State & Data**\n- Lift state up to context or parent components
when needed; avoid duplicating state in multiple components.\n- Favor
composition over inheritance for reusable UI logic.\n\n**Error Handling**\n-
Gracefully handle null/undefined props (use optional chaining and default
props).\n- Display user-friendly error messages for failed API calls, using a
common `<ErrorBoundary>` where appropriate.\n"
    },
    {
      "stack": "fastapi",
      "filename": "fastapi-backend.mdc",
      "content": "--\ndescription: Python FastAPI backend best practices and
architecture\nglobs:\n  - \"app/**\"\n  - \"**/*.py\"\nalwaysApply: true\n--
\n**Architecture & Structure**\n- Follow a layered structure: `app/main.py`
for startup, separate **routers** for each feature, and dedicated `schemas/`
(Pydantic models) and `services/` modules [18] .\n- Use **dependency injection**
via FastAPI `Depends` for shared resources (database sessions, auth).
\n\n**Type Safety & Models**\n- Define request/response models with
**Pydantic**; use type hints on all endpoints and functions [19] .\n- Validate
inputs with Pydantic (e.g. field constraints) and use FastAPI’s automatic
JSON Schema generation for docs [19] .\n\n**Coding Conventions**\n- Prefer async
def for endpoints and DB calls (to leverage non-blocking I/O) [20] .\n- Handle
errors with FastAPI’s HTTPException (e.g. `HTTPException(status_code=400,
detail=\"...\")`) [21] .\n- Log important events and errors (use Python’s
logging library configured for JSON output in production).\n- Adhere to **PEP
8** style guidelines for all Python code (use a linter like flake8).
\n\n**Performance & Security**\n- Use FastAPI’s background tasks for any
long-running jobs (to avoid blocking request handling) [22] .\n- Enable CORS
only for allowed origins and use `Middleware` for cross-cutting concerns
(e.g. timing, error transforming).\n- Never store secrets in code; use
environment variables and a config management system.\n- Implement unit and
integration tests for each router (ensure 80%+ coverage)."
    },
    {
      "stack": "node-express",
      "filename": "node-express-guidelines.mdc",
      "content": "--\ndescription: Node.js Express server guidelines (REST
API, security, style)\nglobs:\n  - \"server/**\"\n  - \"**/*.js\"\n  - \"**/
*.ts\"\nalwaysApply: false\n--\n**Architecture**\n- Use Express Router to
structure endpoints by domain (e.g. `/routes/userRoutes.js`, `/routes/
productRoutes.js`). Mount routers in a central `app.js`.\n- Separate
concerns: route definitions call controller functions in `controllers/`,
business logic in `services/`, DB queries in `models/`.\n\n**Coding Best
Practices**\n- Use modern JS/TS syntax: prefer **async/await** for async
```

```
calls, avoid callbacks. Handle promise rejections with try/catch.\n- Include
**input validation** for every route (use middleware like celebrate/Joi for
schema validation) [23] .\n- Implement centralized error handling middleware
(`app.use(errorHandler)`) to catch exceptions. Return structured JSON errors
(with HTTP status codes).\n- Enforce code style with a linter (e.g. ESLint
Airbnb config) and format with Prettier.\n\n**Security**\n- Sanitize all
inputs to prevent injection attacks (use parameterized queries or an ORM for
DB access).\n- Use **Helmet** middleware to set secure HTTP headers, and
enable CORS only for trusted domains.\n- Store secrets (DB credentials, API
keys) in environment variables; never commit them.\n- Implement
authentication and authorization checks on protected routes (e.g. JWT
verification middleware for `/api/private/*`).\n- Avoid using `eval` or
executing OS commands from user input (command injection risk).
\n\n**Performance & Scaling**\n- Use clustering or PM2 in production to
exploit multi-core CPUs (or prefer async non-blocking operations to handle
concurrency).\n- Cache frequently requested data in memory (or use Redis) to
reduce DB load.\n- Graceful shutdown: handle SIGINT/SIGTERM to close server
and DB connections on exit.\n"
    },
    {
      "stack": "other",
      "filename": "secure-coding-standards.mdc",
      "content":
"--\ndescription: General security and quality rules for all projects\nglobs:
\n  - \"**/*\"\nalwaysApply: true\n--\n**Input Validation & Output
Encoding**\n- **Validate all inputs** from users or external APIs. Reject or
sanitize inputs that don’t meet expectations (length, format, type) to
prevent injections (SQL, NoSQL, or command injection) [24] .\n- **Escape HTML/
XML output** to prevent XSS. Never insert untrusted data into responses
without encoding.\n\n**Authentication & Secrets**\n- Use strong
authentication: prefer standard protocols (OAuth 2.0/OIDC for user logins)
instead of DIY methods.\n- Never hard-code secrets (keys, passwords) in code
 [24] . Utilize environment variables or secret management services. The AI
should avoid printing or logging sensitive secrets.\n- Enforce
**authorization checks** on every restricted action (e.g., verify user roles/
permissions for admin routes). Ensure **access control** is checked server-
side (CWE-862).\n\n**Error Handling**\n- Don’t reveal stack traces or
internal error details to users. Log detailed errors server-side, but return
generic messages to clients (to avoid leaking info).\n- Implement global
exception handlers to catch and respond to unexpected errors gracefully
(avoid app crashes).\n\n**Coding Standards**\n- Avoid undefined behavior and
risky functions. E.g., in C/C++ (if used) avoid `gets()`; in JavaScript,
prefer safe libraries over using `eval`.\n- Follow language-specific best
practices (e.g., use parameterized queries with prepared statements in SQL,
use ORM for DB access to prevent SQL injection (CWE-89)).\n- Regularly update
dependencies to pick up security fixes; have the agent assist in checking for
known vulnerabilities (use `npm audit`/`pip audit`).\n"
    }
  ],
  "prompts": [
    {
      "name": "/plan",
      "mode": "composer",
      "content": "Think through the task in detail and outline a plan before
coding. First, **list all the steps** required to implement the requested
feature or fix as a checklist (e.g. Step 1, Step 2, ...), referencing
specific files or functions. *Do not write any code yet.* Once I approve the
plan, proceed to execute the steps one by one, checking them off as
completed."
    },
    {
      "name": "/tdd",
      "mode": "composer",
      "content": "We will use Test-Driven Development. Based on the feature
description or bug report provided, **write a failing test case** (or cases)
that captures the expected behavior. Use our existing testing framework and
include any necessary setup/teardown. Make sure the test fails initially
(Red). Then, **write the minimum code** to make the test pass (Green) while
following our coding standards. Finally, if appropriate, **refactor** the
code for clarity or efficiency without breaking the test (Refactor). Provide
the diff for the implementation and ensure the test passes."
    },
    {
      "name": "/fix-from-trace",
      "mode": "composer",
      "content":
"Given the following error trace or bug description, analyze the root cause
and propose a fix. **Identify the offending code** and explain why it’s
failing. Then provide a patch as a unified diff that resolves the issue.
Ensure the fix aligns with our style and doesn’t introduce new errors (the
agent should run relevant tests after applying the fix). If additional
context is needed (e.g., content of a file), ask for it rather than
guessing."
    },
    {
      "name": "/pr",
      "mode": "composer",
      "content": "Prepare a pull request description and commit message for
the changes just made. The commit message should follow **Conventional
Commits** format (e.g., `feat:`, `fix:`, etc.) summarizing the change. The PR
description should include:\n- **What** changes were made, and **why** (the
problem it solves).\n- Any relevant context or link to issue/ticket.\n- How
to test or verify the changes.\nFormat the description in Markdown, use
bullet points or paragraphs for clarity, and ensure it provides reviewers
enough information. Do not auto-merge – just output the proposed title and
description."
    },
    {
      "name": "/refactor",
      "mode": "composer",
      "content": "Refactor the selected code or module to improve
readability, maintainability, and performance **without changing its external
```

```
behavior**. Begin by describing the intended improvements (e.g., simplify
logic, remove duplication, improve naming, break into smaller functions).
Then apply the changes. Ensure all tests continue to pass. Provide the diff
of the refactored code and briefly explain the key changes (as code comments
or summary). Focus on clean code principles (SOLID, DRY, etc.) and our
project’s style rules during refactoring."
    },
    {
      "name": "/deploy-staging",
      "mode": "composer",
      "content": "Automate deployment for the application to the staging
environment. Generate the necessary configuration or scripts (for example, a
GitHub Actions workflow YAML, Dockerfile, and docker-compose or a Cloud build
config) to build and deploy the current project to a staging server. The
deployment process should include:\n- Running tests and linting (CI) before
deploying.\n- Building the application (and Docker image if applicable).\n-
Deploying to the staging environment (e.g., pushing to a staging branch or
triggering a deploy service API).\nInclude placeholders for any secrets (do
not hardcode credentials; reference environment variables). Ensure the
workflow is idempotent and includes notifications on success/failure. Output
the configuration files and any relevant instructions."
    }
  ],
  "model_matrix": [
    {
      "task": "Small code generation (boilerplate, simple functions)",
      "model": "OpenAI GPT-3.5 Turbo",
      "params": {
        "max_tokens": 2048,
        "temperature": 0.7
      },
      "expected_cost_latency": "Fast and cheap – typically <2 seconds per
request, costing around $0.002 per 1K tokens. Ideal for quick tasks and
boilerplate code [25] , but may produce simpler solutions and occasionally skip
details due to its limited context understanding."
    },
    {
      "task": "Complex logic or multi-file refactor",
      "model": "OpenAI GPT-4 (or GPT-4o)",
      "params": {
        "max_tokens": 8192,
        "temperature": 0.2
      },
      "expected_cost_latency": "Moderate speed (~5-10 seconds) and higher
cost (~$0.06 per 1K tokens). Provides reliable, coherent outputs for
intricate logic and larger refactors [26] . GPT-4 is strong at reasoning and
following instructions, but has a smaller context than Claude 4.1. Good for
critical code where quality matters more than speed."
    },
    {
      "task": "Large context understanding (architecture Q&A, cross-file
analysis)",
      "model": "Anthropic Claude 4.1 (Opus)",
      "params": {
        "max_tokens": 1000000,
        "temperature": 0.5
      },
      "expected_cost_latency": "Slower (~15+ seconds for large prompts) and
expensive (supports million-token context, costs proportionally high). Excels
at **holistic codebase reasoning** – it can ingest entire subsystems and
provide insights or refactor plans as if a senior architect wrote them [27] .
Claude’s strengths are long-form consistency and focus, but it may refuse
ambiguous requests or wander on tangents if not guided [28] ."
    },
    {
      "task": "Codebase search and quick fixes (Agent Auto mode)",
      "model": "Cursor Auto (dynamic model selection)",
      "params": {
        "auto": true
      },
      "expected_cost_latency": "Varies – Cursor will pick a model for
reliability. In practice, Auto uses fast models for small edits and switches
to more powerful ones for complex tasks [29] . This yields a balanced outcome:
minimal wait for trivial changes, and more thought for challenging ones. Cost
stays within subscription limits (fast requests quota) for normal use [30] [31] .
Use Auto when unsure which model fits best or to avoid manual model
juggling."
    },
    {
      "task": "Documentation queries and examples integration",
      "model": "Google Gemini 2.5 Pro",
      "params": {
        "max_tokens": 2000000,
        "temperature": 0.3
      },
      "expected_cost_latency":
"Specialized – handles extremely large context (up to ~2M tokens) for pulling
in documentation or entire code files. Latency is moderate (~8-12 seconds)
and cost is high per call. Gemini is effective for **documentation Q&A and
codebase-wide analysis** – e.g., asking for all uses of a function across a
huge repo [32] [33] . Use it sparingly for research tasks where its context length
shines."
    }
  ],
  "kpis": {
    "definitions": [
      "**First-pass success rate** – percentage of agent tasks completed
without human intervention or re-runs. For example, if out of 10 requested
features/fixes the agent finished 7 with all tests passing on the first
attempt, the success rate is 70%. This measures how often the agent “gets it
right” initially.",
      "**Average iterations per task** – the mean number of plan-execute
```

```
cycles the agent goes through for each task. An iteration includes planning,
coding, testing, and possibly a self-review. Fewer iterations mean the agent
converged on a solution quickly; a high number may indicate thrashing or
misunderstandings that needed correction.",
      "**Time to completion (TTF)** – the elapsed time from when an agent
task is started to when it is completed (all tests passing and code
reviewed). Tracked in minutes. This KPI helps evaluate latency: e.g., simple
fixes might be 1-2 minutes, whereas a multi-file refactor might take 10+
minutes of agent work (including tool runs). Shorter TTF means faster
delivery.",
      "**Average diff size** – the average number of lines of code changed
per agent task (addition + deletion count). This indicates the scope of
changes. We expect small, incremental diffs (e.g., <50 lines) for most tasks
due to our safe-edit protocol. A much larger average might point to the agent
taking on too much at once, increasing risk.",
      "**Cost per task** – approximate compute cost in API usage for each
task. For example, measured in tokens or dollars spent (if using pay-as-you-
go). This can be derived from model token usage logs: e.g., a complex task
might consume 20K tokens (~$0.04 on GPT-4). Monitoring this helps optimize
when to use cheaper models or Auto mode. It’s expressed as $/task on
average."
    ],
    "collection": "We will instrument the agent’s runs via logging and CI
hooks. Each agent invocation emits structured logs (JSON) including
timestamps for each step, the plan and iteration count, and test results. A
custom **Cursor Agent Monitor** script parses these logs at task end to
compute metrics. For example, it can count iterations (each time the agent
marks a plan step complete) and measure duration from start to finish. We
also integrate with the test runner: after each task, the test suite results
are recorded to see if it passed on first attempt or needed fixes (for
*first-pass success rate*). Diff size can be obtained by analyzing the git
diff produced by the agent’s commit (we’ll have the agent or a git hook label
each commit with task ID and count lines changed). Cost is tracked by
capturing model usage: Cursor provides token count or model API cost info per
request in verbose logs (or we use the Model’s API usage metrics if
available). These logs are aggregated in a dashboard (e.g., a small web UI or
even a markdown report) for each project, so we can see trends in agent
performance over time. In summary, KPIs are collected automatically through
agent hooks and logging, and summarized in the CI/CD pipeline after agent-
driven updates."
  },
  "security": {
    "secrets_handling": "All secrets are strictly handled outside the AI’s
direct reach. We mark secret files (like `.env` or config keys) in
**.cursorignore** so the agent cannot read them [10]. During operations, any
credentials needed (for e.g. deployment) are provided via environment
variables or vault integration – never typed into the prompt. We enable
Cursor’s **Hooks** feature to automatically redact any secret-like patterns
if the agent tries to output them (for example, if a string looks like an API
key, the hook replaces it with `<REDACTED>` before it reaches the chat) [15].
The agent is instructed (via rules and system prompts) not to log or expose
```

```
sensitive info. When running commands, it uses interpolated environment
variables (supported in MCP configs) instead of inlining secrets [34] . For
instance, our deployment command uses `$AWS_TOKEN` rather than the actual
token. We also review the agent’s suggestions for any accidental leakage. By
combining ignore rules, hooks, and careful prompt design, we ensure the agent
works with secrets safely—using them when needed (e.g., in a config file
template) but never revealing them in conversation or version control.",
    "cwe_checklist": [
      "CWE-79 (Cross-Site Scripting): Ensure all user input reflected in UI
is escaped or sanitized [24] . In React, avoid `dangerouslySetInnerHTML` with
unsanitized data.",
      "CWE-89 (SQL Injection): Always use parameterized queries or ORM
methods for database access. Never concatenate untrusted input into SQL
queries.",
      "CWE-522 (Missing Encryption): Use TLS for all network calls. Store
sensitive data (passwords, tokens) only in hashed or encrypted form (e.g.,
bcrypt for passwords).",
      "CWE-306/862 (Missing Authentication/Authorization): Enforce auth
checks on every restricted action. The agent should include permission checks
(e.g. verifying JWT and role) in generated code for protected endpoints.",

"CWE-798 (Hardcoded Secrets): No hardcoded credentials or keys in code. The
agent should use env variables or config files for secrets and mention if it
detects any in code for removal [24] .",
      "CWE-120/787 (Buffer Overflow): (If using lower-level languages) the
agent should avoid unsafe functions (`gets`, `strcpy` in C) and prefer
bounded alternatives (`fgets`, `strncpy`). Though our stacks are memory-safe
(Python, JS), this is noted for any C/C++ components.",
      "CWE-400 (Resource Exhaustion): Ensure loops, recursion, or memory
usage in generated code are bounded. The agent should, for example, guard
against reading extremely large files into memory or unbounded while loops.",

"CWE-94 (Code Injection): The agent should never construct new code via eval
from user input. In Node, avoid `eval` or dynamic `require` with user data.
In Python, avoid `exec` on untrusted strings."
    ],
    "pot_sandbox": "We operate the agent in a \"Plan-of-Thought\" sandbox to
constrain its actions. **Agent commands run in an isolated environment** –
for example, a Docker container or VM that mirrors the development setup but
has limited permissions. This means if the agent executes malicious or
destructive commands, it won’t harm developer machines or production data. We
configure Cursor’s agent terminal to use a Docker context (via the CLI
integration) so that any `npm install` or filesystem changes occur in that
container. Additionally, using **Hooks**, we intercept critical operations:
e.g., if the agent tries to delete a file outside the project or make system-
level changes, the hook either blocks it or requires explicit user approval
 [15] . The agent’s \"thoughts\" (the sequence of planned actions) are thus
effectively sandboxed— it can propose an action, but our guardrails ensure it
executes in a safe playground. For potentially dangerous tasks (like running
database migrations or deployment scripts), we have a dry-run mode the agent
must use first, and output is reviewed. This PoT sandbox approach lets the
```

```
agent be ambitious in planning and tool use, while we contain the impact to a
safe environment. After confirming the changes are valid and safe, they can
be applied to the real system (often via a controlled merge or deploy process
rather than directly from the agent)."
  },
  "failures": [
    {
      "name": "Stuck in Loop (Agent repeats without progress)",
      "symptoms": [
        "The agent keeps repeating the same plan step or message without
moving forward.",
        "Agent says it will do something but then re-states the plan or
problem again (commonly during the review or planning phase).",
        "Time passes with multiple identical or very similar outputs, and no
new code changes."
      ],
      "fix_prompts": [
        "Interrupt the agent and clarify the next step. For example: *\"You
have repeated the plan. Please proceed with implementing Step X now.\"* This
direct instruction can jolt it out of the loop.",
        "If the loop persists, try rephrasing the request or breaking the
task into a smaller chunk. You might say: *\"Ignore the previous plan for a
moment and just focus on accomplishing [specific sub-task].\"*",
        "Use the `/summarize` command to condense the conversation, then try
the plan again. Summarizing context can reduce token load and sometimes
clears whatever the model was stuck on.",
        "As a last resort, restart the agent in a fresh session with the key
context (files and rules) reattached, then give a concise directive for the
next action. This often stops whatever recursive thought pattern it was in."
      ]
    },
    {
      "name": "Partial Edits Not Applied",
      "symptoms": [

"The agent claims to have made multiple changes, but only some appear in the
code.",
        "It says \"Applied changes to files A, B, C\", but upon inspection
file B or C is unchanged.",

"Often occurs when the agent attempts many simultaneous file edits in one go
– some edits vanish or overwrite each other."
      ],
      "fix_prompts": [
        "Instruct the agent to apply changes one file at a time. E.g.:
*\"Let's do this step-by-step. First, update file A as discussed. (After it's
done) Now update file B...\"*. This ensures each edit is committed.",
        "Ask the agent to show a unified diff of the intended changes. This
helps identify which part didn't apply. Then you can say: *\"The changes to
file B didn't go through, please redo them as per the diff.\"*",
        "If using an older Cursor version with a known multi-edit bug,
upgrade to the latest version where this is fixed [35]. Mention to the agent that it should focus on one save at a time. You can prompt: *\"Save each file after editing before moving to the next.\"*",
        "Manually apply any missing changes if urgent, or use the agent in inline mode (file-by-file) instead of agent mode for bulk edits. Then continue with agent mode for the remaining tasks."
      ]
    },
    {
      "name": "Context Drift / Forgotten Requirements",
      "symptoms": [
        "After a long conversation or many edits, the agent’s output no longer addresses the original request.",
        "The agent introduces unrelated changes or regresses earlier decisions (e.g., reverts a naming convention or uses a disallowed library) that were settled before.",
        "It asks questions or makes changes that indicate it 'forgot' some context from earlier in the session."
      ],
      "fix_prompts": [
        "Remind the agent of the key requirements and context. E.g.: *\"Remember: we must use X library and follow Y pattern as established earlier.\"*. Often explicitly restating the requirement realigns the model.",
        "Use the `@Notepad` or a summary: *\"Here’s a summary of what we’ve done and decided: ...\"*. By re-injecting a concise summary of prior context, the agent can recover the thread [36].",
        "If the conversation is very long, consider using `/summarize` to compress older parts of the chat [37], or start a new session with critical files and a recap of the goal. Then continue – this reduces token load and confusion.",
        "Set intermediate checkpoints: after completing a subset of tasks, have the agent summarize progress and the remaining to-do list. Prompt: *\"Summarize what has been done and what’s left.\"* This ensures alignment before moving on."
      ]
    },
    {
      "name": "Tool Failure or Environment Error",
      "symptoms": [
        "The agent runs a terminal command (like tests or builds) and it hangs indefinitely, or returns an error the agent doesn’t automatically address.",
        "E.g., agent says *\"Running tests...\"* but nothing happens, or a command like `npm install` fails due to a lockfile issue and the agent output stops.",
        "The agent might appear to ignore the failure and not progress."
      ],
      "fix_prompts": [
        "Intervene by providing feedback on the failure. E.g.: *\"The tests command seems to have hung. Perhaps there's an interactive prompt or it’s waiting for input.\"*. This can prompt the agent to try an alternative (like
adding a `--no-interactive` flag or checking logs).",
        "If a command consistently hangs (a known issue pre-1.6), update
Cursor (the 1.6 release fixed many terminal hanging issues [38] ). In-session,
you can manually kill the process (if possible) and tell the agent: *\"The
last command hung, try a different approach (or skip it).\"*",
        "Prompt the agent to handle errors: *\"If the build fails, capture
the error output and address it.\"*. This makes the agent actively respond to
tool failures rather than go silent. For example, if `npm install` fails, the
agent could run `npm cache clean` or explain the error.",
        "Run the troublesome command outside agent (locally) to see output,
then feed the error back into the agent. *\"I ran the tests and got this
error: ... Please fix it.\"*. By giving the agent the exact error, you get it
unstuck and back on track to fix the issue."
      ]
    }
  ],
  "references": [
    {
      "id": "ref1",
      "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": "ref2",
      "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": "ref3",
      "title": "Slash commands, summarization, and improved Agent terminal
(Cursor Changelog 1.6)",
      "url": "https://cursor.com/changelog/1-6",
      "type": "doc",
      "year": 2025
    },
    {
      "id": "ref4",
      "title": "Agent Autocomplete, Hooks, and Team Rules (Cursor Changelog
1.7)",
      "url": "https://cursor.com/changelog/1-7",
      "type": "doc",
      "year": 2025
    },
    {
      "id": "ref5",
      "title": "Choosing the Right Model in Cursor – Frontend Masters Blog",
      "url": "https://frontendmasters.com/blog/choosing-the-right-model-in-cursor/",
      "type": "post",
      "year": 2025
    },
    {
      "id": "ref6",
      "title": "Benchmarking Large Codebase Search with Cursor, Windsurf, Claude Code, Copilot, Codex, Augment, and Jolt",
      "url": "https://www.usejolt.ai/blog/large-codebase-search-benchmark",
      "type": "benchmark",
      "year": 2025
    },
    {
      "id": "ref7",
      "title": "Cursor vs Sourcegraph Cody: Embeddings and Monorepo Scale",
      "url": "https://www.augmentcode.com/guides/cursor-vs-sourcegraph-cody-embeddings-and-monorepo-scale",
      "type": "post",
      "year": 2025
    },
    {
      "id": "ref8",
      "title":
"PatrickJS/awesome-cursorrules (GitHub repository of community Cursor rules)",
      "url": "https://github.com/PatrickJS/awesome-cursorrules",
      "type": "repo",
      "year": 2025
    }
  ],
  "meta": {
    "time_window": "2025-06-01 to 2025-09-30",
    "source_mix": {
      "docs": 2,
      "repos": 1,
      "case_studies": 1,
      "benchmarks": 1
    },
    "confidence_notes":
"All recommendations were drawn from mid-2025 primary sources, including official Cursor documentation and real user case studies, which increases our confidence in their accuracy. For example, the plan→execute workflow and rules usage are backed by Cursor’s own guides and a detailed Medium post [1] [3]. The impressive productivity gains and pitfalls (like agent overreach requiring rollback) come directly from a Salesforce engineering case study [2], lending credibility to our safety protocols. We cross-validated practices like TDD with community discussions (e.g. using tests to keep the agent on
track [13] ) and they align with known best practices in AI-assisted coding.
Where exact data was unavailable (such as the meaning of 'PoT sandbox' which
isn’t a formal term in Cursor docs), we inferred a reasonable interpretation
based on hooks and sandboxing concepts introduced in recent updates [15] .
Potential gaps: specific slash command internals (e.g., `/deploy-staging`)
were not explicitly documented in sources, so we extrapolated those from
typical workflows. Trade-off metrics (model speed, cost) are approximated
from referenced blog insights rather than official numbers [25] . Overall, the
sources consistently support the strategies presented, and we’ve noted
assumptions where we had to interpret context (marked by terms like “we
inferred” or “likely”). No conflicting evidence was found in the given
timeframe, but as with any rapidly evolving tool, some details (like model
versions or pricing) could change beyond September 2025."
  }
}
```

---

[1] [3] [5] [9] [10] [11] [36] 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

[2] [6] [14] How Cursor AI Cut Legacy Code Coverage Time by 85%
https://engineering.salesforce.com/how-cursor-ai-cut-legacy-code-coverage-time-by-85/

[4] GitHub - PatrickJS/awesome-cursorrules: Configuration files that enhance Cursor AI editor experience with custom rules and behaviors
https://github.com/PatrickJS/awesome-cursorrules

[7] [8] [32] [33] Cursor vs Sourcegraph Cody: embeddings and monorepo scale - Augment Code
https://www.augmentcode.com/guides/cursor-vs-sourcegraph-cody-embeddings-and-monorepo-scale

[12] Benchmarking Large Codebase Search with Cursor, Windsurf, Claude Code, Copilot, Codex, Augment, and Jolt | Jolt AI
https://www.usejolt.ai/blog/large-codebase-search-benchmark

[13] [23] All you need is TDD. : r/cursor
https://www.reddit.com/r/cursor/comments/1iq6pc7/all_you_need_is_tdd/

[15] Agent Autocomplete, Hooks, and Team Rules · Cursor
https://cursor.com/changelog/1-7

[16] [17] [24] Top Cursor Rules for Coding Agents
https://www.prompthub.us/blog/top-cursor-rules-for-coding-agents

[18] [19] [20] [21] [22] .cursorrules
https://github.com/PatrickJS/awesome-cursorrules/blob/cfdccb0332370077e7cb20a9649429dc9d85ae20/rules/python-fastapi-cursorrules-prompt-file/.cursorrules

[25] [26] [27] [28] [29] Choosing the Right Model in Cursor – Frontend Masters Blog
https://frontendmasters.com/blog/choosing-the-right-model-in-cursor/

[30] [31] Cursor AI Cost Analysis: Evaluating the Value for Developers
https://www.sidetool.co/post/cursor-ai-cost-analysis-evaluating-the-value-for-developers/

[34] [37] [38] Slash commands, summarization, and improved Agent terminal · Cursor
https://cursor.com/changelog/1-6

35 Multiple edits has stopped working - Bug Reports - Cursor Forum
https://forum.cursor.com/t/multiple-edits-has-stopped-working/132136