Deployable Architectures for Production-Grade Coding Agents in Cursor\n\n## I. Foundational Architecture: Configuring the Agent's Environment\n\nThe successful deployment of a production-grade coding agent begins not with a prompt, but with a meticulously configured environment. This foundational architecture is designed to establish a predictable, controllable, and context-aware system where agent behavior is declaratively managed and its capabilities are explicitly defined. The evolution of the Cursor platform between June and September 2025 reveals a clear trajectory towards treating agent configuration with the

same rigor as infrastructure-as-code. This section details the hierarchical rules engine, advanced context scoping strategies, and the extensible tooling protocol that form the bedrock of a reliable autonomous system.\n\n### 1.1 The Rules Engine Hierarchy: Declarative Control via Configuration-as-Code\n\nThe governance of an AI agent's behavior in a complex software project cannot be left to the vagaries of a single prompt. A multi-layered system of rules provides the necessary structure to enforce standards, guide actions, and ensure consistency at different scopes. This approach, which treats agent behavior as a version-controlled, configurable asset, is a direct parallel to the

maturation of configuration-as-code principles in the DevOps domain. The trajectory of Cursor's rules engine, from a single configuration file to a modular, directory-based system, underscores this parallel, suggesting that managing agent behavior requires the same principles of versioning, modularity, and declarative control.

13

Security Mandates: "Under no circumstances will the agent commit API keys, passwords, or other secrets to version control. All sensitive data must be managed through environment variables or a designated secrets management service."\n* **Commit Message Formatting:** "All git commit messages must adhere to the Conventional Commits specification. The agent is responsible for generating a compliant message that accurately summarizes its changes."\n* **Legal and Licensing Compliance:** "The agent must not introduce code with incompatible licenses (e.g., GPL) into the codebase. All new dependencies must be vetted against the organization's approved license list."\n\nBy establishing these rules at the team level, organizations ensure that all agent activity, regardless of the project, adheres to fundamental operational and security baselines. This global governance is the first and most critical step in mitigating the risks associated with autonomous code generation.\n\n#### Repository-Level Rules (.cursor/rules/*.mdc)\nThe core of project-specific behavior is defined at the repository level. The evolution from a single .cursorrules file to a directory of modular rule files (.cursor/rules/) is a critical adaptation for managing complexity, especially in monorepositories.[2] This structure allows for the creation of domain-specific instruction sets that apply to different parts of the codebase, governed by glob patterns specified within the rule files themselves.[5] This modularity is paramount for projects that contain disparate technology stacks, such as a React frontend and a Python backend, which require different coding standards, architectural patterns, and testing methodologies.\n\nThe

awesome-cursorrules repository, a community-maintained collection of best practices, provides excellent templates for production-grade rule files.[6] For a typical monorepo, the

.cursor/rules/ directory might contain:\n* frontend.mdc: Specifies rules for the React/TypeScript portion of the app, including component structure, state management patterns (e.g., Zustand), and adherence to UI library conventions like shadcn/ui.[5]\n*

backend.mdc: Defines standards for the FastAPI backend, such as dependency injection patterns, Pydantic model usage, and database interaction conventions.[4]\n*

ci-cd.mdc: Contains instructions for interacting with the CI/CD pipeline, such as how to correctly format GitHub Actions workflow files or GitLab CI configurations.[7]\n*

documentation.mdc: Enforces standards for updating README.md files, generating code comments, or maintaining project documentation.\n\nThis separation of concerns allows the agent to dynamically load the most relevant instructions based on the file it is currently editing, dramatically improving the quality and consistency of its output. These rule files should be committed to version control, making agent behavior as reviewable and auditable as the application code itself.\n\n#### Slash Commands (.cursor/commands/*.md)\nIntroduced in version 1.6, Slash Commands are user-facing, reusable prompts that function as executable playbooks stored in the .cursor/commands/ directory.[3] They represent the most immediate layer of control, allowing developers to invoke complex, multi-step workflows with a simple command. While repository rules define

how the agent should behave passively, slash commands define *what* the agent should do actively.\n\nThey are ideal for encapsulating and standardizing recurring engineering tasks, thereby democratizing complex workflows and reducing manual effort. A standard library of slash commands for a production environment would include:\n* /lint: "Execute the project's linter (eslint --fix). Analyze the output and apply all auto-fixable corrections. For errors that cannot be auto-fixed, list them for my review."\n* /test: "Run the entire test suite (npm test). If any tests fail, analyze the failure logs, identify the root cause, and implement a fix. Repeat this process until all tests pass."\n* /pr: "Generate a pull request. First, inspect the git diff --staged. Write a PR description that summarizes the changes and links to the relevant Linear issue. Use a Conventional Commit message for the PR title. Then, create the pull request using the GitHub integration."\n* /refactor: "Please refactor the selected code. Your priorities are to improve readability, reduce cyclomatic complexity, and ensure adherence to the SOLID principles defined in our backend.mdc rules file."\n\nThis hierarchical, file-based, version-controllable system for managing agent behavior is not merely a convenience; it is a necessity for production use. It provides the structure required to scale agentic workflows across teams and projects while maintaining high standards of quality, security, and consistency.\n\n### 1.2 Advanced Context Scoping Strategies (RAG)\n\nThe performance of a Large Language Model is fundamentally constrained by the quality of its context. In the domain of coding, providing the agent with precise, sufficient, and cost-effective context is

the primary mechanism for minimizing drift, preventing hallucination, and ensuring generated code is relevant and correct. This process, a form of Retrieval-Augmented Generation (RAG), is managed in Cursor through a combination of automated tools and curated artifacts.\n\nAn effective RAG strategy combines automated, tactical context retrieval with manually curated, strategic context. The former provides the agent with the immediate code it needs to perform a task, while the latter provides the architectural and intentional guardrails that ensure the task is performed correctly within the project's broader goals. This hybrid approach is the most effective known method for mitigating the primary failure mode of long-running agentic tasks: contextual drift, where the agent gradually loses sight of the original intent.[9]\n\n#### Strategic Use of

@ Symbols\nCursor's @ symbol provides a powerful interface for dynamically injecting context into a prompt. However, each method comes with trade-offs in terms of precision, token cost, and relevance.[11]\n*

@Files & @Folders: This is the most precise method for providing context, allowing the user to specify exact files or directories. It is ideal for targeted refactoring or when working on a feature that spans a small, known set of files. Its primary drawback is its high token cost, which can quickly consume the model's context window.\n* @Codebase: This command leverages Cursor's codebase indexing to perform a semantic search and retrieve relevant snippets from across the entire project.[11] It is highly effective for answering broad questions like "Where is user authentication handled?" or for tasks where the relevant files are not immediately known. Its effectiveness, however, is entirely dependent on the quality and freshness of the index. Users must ensure the index is kept up-to-date to avoid the agent working with stale information.[10]\n*

@Web: This allows the agent to search the internet, providing access to the latest documentation, library versions, or solutions to common errors.[11] It is invaluable when working with new technologies or troubleshooting obscure bugs. However, it also introduces external, potentially unreliable, or even insecure information into the context. Its use should be governed by rules that instruct the agent to prioritize official documentation over forum posts.\n*

@Docs: This feature allows teams to add their own private or project-specific documentation to Cursor's knowledge base.[11] This is arguably the most critical context provider for enterprise use, as it allows the agent to be grounded in the team's internal libraries, APIs, and architectural decision records (ADRs), which are not available on the public web.\n\n#### The

llm-context.md Pattern\nAn emerging best practice from the developer community is the creation of a root-level llm-context.md file.[14] This document serves as a persistent, high-level "constitution" or "mission brief" for the project. Unlike rule files, which contain specific,

low-level instructions, the

llm-context.md file provides strategic, architectural context. It is the most effective tool for combating the "context forgetfulness" that plagues long-running agent sessions.[9]\n\nA comprehensive

llm-context.md should include:\n1. **Project Purpose:** A concise, one-paragraph summary of what the application does and for whom.\n2. **Technology Stack:** An explicit list of major frameworks, libraries, and languages used (e.g., "Frontend: Next.js 14 with TypeScript, Tailwind CSS, and Zustand. Backend: Python 3.11 with FastAPI and PostgreSQL.").\n3. **Architectural Principles:** High-level patterns the agent must follow (e.g., "This is a modular monorepo. All shared logic must reside in the /packages directory.", "We use React Server Components by default.", "All database access must go through the repository layer; never call the ORM directly from an API route.").\n4. **Critical "Do-Not-Dos":** A list of anti-patterns to avoid (e.g., "Do not introduce new dependencies without approval.", "Do not use any in TypeScript.", "Do not commit directly to the main branch.").\n\nAt the beginning of a new task or after a context reset, the developer's first action should be to instruct the agent: "Read and internalize the contents of @llm-context.md before you begin." This acts as a "system prompt anchor," ensuring the agent's core mission and constraints are always at the forefront of its reasoning process.\n\n#### Managing Context Window Limits\nThe finite context window of LLMs is a hard physical constraint. As a conversation progresses, this window can become cluttered with irrelevant information, degrading performance. Cursor provides two key mechanisms for managing this:\n* **On-Demand Summarization:** The /summarize slash command, introduced in version 1.6, allows the user to instruct the agent to condense the current conversation history into a concise summary, freeing up tokens for new information.[3] This is useful when pivoting to a new task within the same chat session.\n*

Recent Changes Awareness: The agent can now see changes made to files between user messages, meaning it doesn't need to be explicitly re-told about edits it has just made.[2] This reduces redundant context.\n\nHowever, the most robust technique for managing context remains the "controlled context reset".[10] When an agent begins to drift or perform poorly, starting a new chat session is the most effective remedy. This action flushes the short-term conversational memory while preserving the long-term, high-value context provided by the codebase index and the

llm-context.md file. This suggests an operational pattern where agent sessions are treated as short-lived and task-specific, preventing the accumulation of contextual noise.\n\n### 1.3 Tooling and Extensibility with Model Context Protocol (MCP)\n\nTo achieve true end-to-end automation, a coding agent must be able to interact with the world beyond the local filesystem. It needs to create pull requests, update issue trackers, and trigger deployments. The Model Context Protocol (MCP) is the key enabling technology that transforms the Cursor Agent from a sophisticated code editor into a genuine automated team member capable of participating in the full software development lifecycle.[16]\n\nMCP is a standardized

specification that allows LLMs to discover and interact with external tools and APIs in a structured, predictable manner. Cursor's support for MCP, including features for resource sharing (v1.6) and structured user input elicitation (v1.5), is a cornerstone of its production-grade capabilities.[3]\n\n#### Connecting to Production Services\nIntegrating the agent with the services that orchestrate the development process is essential for building automated workflows. Reproducible examples for key integrations include:\n*

GitHub/GitLab: By connecting the agent to the GitHub or GitLab MCP server, it gains the ability to perform version control operations programmatically.[8] A playbook can instruct the agent to: "1. Create a new branch named

feature/TICKET-123-new-auth-flow. 2. Commit the staged changes with a conventional commit message. 3. Push the branch to the remote origin. 4. Create a pull request targeting the develop branch." This automates the entire code submission process.[17]\n*

Linear/Jira: The Linear integration allows agent runs to be initiated directly from a comment on an issue (e.g., @cursor implement this feature).[3] The agent can then read the issue description for context, perform the work, and automatically link its resulting pull request back to the issue, closing the loop on task management.\n*

Deployment Platforms: MCP servers for platforms like Heroku or services like Endgame allow for the creation of plan→code→test→deploy workflows.[8] A slash command like

/deploy-staging could instruct the agent to merge its changes into the staging branch and then use the appropriate MCP tool to trigger a new deployment, reporting back the status.\n\n#### Custom and Community MCP Servers\nThe power of MCP lies in its extensibility. The growing directory of third-party MCP servers allows agents to connect to a wide range of services, from database management (GibsonAI) to feature flagging (Statsig).[16] Furthermore, organizations can develop their own custom MCP servers to expose internal tools, databases, or microservices to the agent. This allows the agent to be deeply integrated into a company's unique technical ecosystem, enabling it to perform tasks like querying internal metrics databases or updating records in a proprietary CRM system.\n\nThe adoption of MCP signifies a critical shift in the role of AI agents. Without it, the agent is a powerful but isolated tool confined to the developer's machine. With MCP, the agent becomes a node in a larger, interconnected system of development and operations tools. A production-grade agent strategy

must leverage MCP to bridge the gap between local code generation and the external systems that manage the software lifecycle.\n\n## II. The Execution Core: Prompts, Playbooks, and Workflows\n\nWith the foundational architecture configured, the focus shifts to execution: the specific instructions and operational procedures that drive the agent to perform complex engineering tasks. This section provides the master prompts, reusable playbooks, and recovery workflows necessary to create a robust, repeatable, and

self-correcting system. The goal is to move beyond simple, one-shot commands and establish a disciplined interaction model that guides the agent toward predictable, high-quality outcomes.\n\n### 2.1 The Autonomous Principal Engineer Prompting Framework\n\nTo elicit behavior characteristic of a senior engineer—methodical, disciplined, and self-sufficient—the agent must be equipped with a master "meta-prompt" or system instruction set. This framework, installed as a global or repository-level rule, establishes the agent's core operational doctrine. It is not a task-specific prompt but a set of guiding principles that govern all its actions.\n\nThis framework synthesizes best practices observed in advanced prompting guides and community-driven frameworks.[19] It is built on three pillars: core principles, structured reasoning, and a safe execution loop.\n\n#### Core Principles\nThe agent's system prompt must explicitly state its foundational principles:\n*

Research-First, Always: "You must never act on assumption. Before writing any code, you will first conduct thorough reconnaissance of the existing codebase to understand the current state, relevant patterns, and potential side effects. You will use tools like @Codebase search and file reading to gather all necessary information."\n* **Extreme Ownership:** "Your responsibility extends beyond the immediate task. You own the end-to-end health and consistency of the system you are modifying. This includes writing or updating tests, updating documentation, and ensuring your changes do not introduce regressions."\n* **Metacognitive Self-Improvement:** "After completing a task, you will reflect on your performance. If you encountered difficulties or made errors, you will analyze the cause and, if appropriate, propose an update to your own rule files to prevent similar mistakes in the future."\n\n#### Structured Reasoning\nTo make the agent's thought process more transparent and predictable, its output should be structured using XML-like tags. This forces the agent to follow a specific reasoning sequence before producing a final answer.[19]\n*

<self_reflection>: The agent should begin every complex task by thinking through the problem. "In this block, you will break down the user's request, identify potential ambiguities, and formulate clarifying questions if the request is underspecified. You will create an internal rubric for what a successful solution looks like."\n* <planning>: The agent must explicitly state its plan of action before execution. "In this block, you will provide a step-by-step plan detailing the files you will create or modify and the commands you will run. You will wait for user approval of this plan before proceeding."\n* <code_editing_rules>: The agent should reiterate the specific coding standards it will follow for the current task, drawn from the relevant .cursor/rules/*.mdc files. "In this block, you will state the guiding principles for your code generation, such as 'All React components will be functional components with TypeScript props' or 'All API endpoints will include Pydantic validation.'"\n\n#### The Plan-Then-Execute Loop\nThe single most important guardrail against agent drift and unintended changes is the enforcement of a strict plan-then-execute loop.[1] The system prompt must mandate this workflow: "You will

always present your plan for approval before making any changes to the filesystem or executing any terminal commands. The user's approval is a required gate to move from the

planning phase to the execution phase." This simple step provides a critical checkpoint for human oversight, preventing the agent from pursuing an incorrect path and wasting time and resources.\n\n#### Safe Edits and Diff Generation\nTo ensure that changes are minimal, targeted, and easily reviewable, prompts must instruct the agent to generate modifications as diffs or patches rather than overwriting entire files. The agent has a native capability to present changes in a diff format, and this should be reinforced in its core instructions.[22] A directive like, "All code modifications must be presented as a minimal diff against the current file content. Do not output the entire file; only show the lines to be added, removed, or changed," ensures that the human reviewer can quickly assess the impact of the agent's work.\n\n### 2.2 Playbook Library for Common Engineering Tasks\n\nBuilding on the foundation of the Principal Engineer framework, playbooks are complete, copy-pasteable prompts for high-value, recurring tasks. They combine the master prompt's principles with task-specific instructions and context. These can be stored as Slash Commands for easy access.[3]\n\n#### Playbook 1: TDD Feature Implementation\nThis playbook guides the agent through a Test-Driven Development workflow, leveraging its ability to execute tests in the integrated terminal.[25]\n\n\n# Task: Implement New Feature via TDD\n\n## Context:\n- Feature Specification: @file:features/new-user-signup.md\n- Relevant Code: @folder:src/server/auth/\n- Project Constitution: @file:llm-context.md\n\n## Instructions:\nYou are to implement the new user signup feature described in the specification. You MUST follow a strict Test-Driven Development (TDD) workflow.\n\n

\n1. **Write a Failing Test:** Create a new test file src/server/auth/signup.test.ts. Write a single integration test that covers the primary success path of the signup flow as described in the spec. This test should fail because the implementation does not exist.\n2. **Run Test:** Execute the test and confirm that it fails as expected.\n3. **Implement Code:** Write the minimum amount of code in src/server/auth/signup.ts required to make the test pass.\n4. **Run Test:** Execute the test again and confirm that it now passes.\n5. **Refactor:** Refactor the implementation code for clarity and adherence to our project's coding standards, ensuring the test continues to pass.\n6. **Add Edge Case Tests:** Add additional tests for failure scenarios (e.g., duplicate email, invalid password) and repeat the implement-test-refactor cycle for each.\n\n\nPresent your plan for approval before beginning.\n\n\n#### Playbook 2: Bug Fix from Stack Trace\nThis playbook instructs the agent to diagnose and fix a bug using logs and a stack trace, a common and high-value workflow.[25]\n\n\n# Task: Diagnose and Fix Production Bug\n\n## Context:\n- Stack Trace & Logs: @file:logs/prod-error-log-12345.txt\n- Potentially Relevant Code: @Codebase("Database connection pooling")\n- Project Constitution: @file:llm-context.md\n\n## Instructions:\nYou are to diagnose and fix the bug detailed in the provided log file. Your approach must be systematic.\n\n

\n1. **Analyze:** Read the stack trace and surrounding logs to form an initial hypothesis about the root cause.\n2. **Reproduce:** If the cause is not immediately obvious, identify the key functions involved. Add detailed logging statements to these functions to get better visibility into the code's execution path and state. Do not make any other changes.\n3. **Propose Fix:** Based on your analysis (and the new logs, if necessary), propose a targeted code change to

fix the bug. Explain *why* your fix will solve the problem.\n4. **Verify:** Describe how you will verify the fix. This should include running relevant tests or describing a manual verification step.\n\n\nPresent your plan for approval. Do not attempt to add logging or fix the code until the plan is approved.\n\n\n#### Playbook 3: Automated PR Generation and Review\nThis playbook, ideally implemented as a /pr slash command, automates the final step of the development workflow.[3]\n\n\n# Task: Create Pull Request\n\n## Context:\n- This command is run after changes have been staged for commit.\n\n## Instructions:\nYou are to create a pull request for the currently staged changes.\n\n

\n1. **Analyze Diff:** Run git diff --staged to understand the changes.\n2. **Generate Commit Message:** Create a single commit message that follows the Conventional Commits specification. The scope should be the primary module affected (e.g., feat(auth):).\n3. **Commit:** Execute the git commit command with the generated message.\n4. **Generate PR Description:** Write a clear and concise pull request description. It should include a 'Summary' section explaining the purpose of the change and a 'Changes' section with a bulleted list of the key modifications.\n5. **Create PR:** Use the GitHub MCP tool to create the pull request, targeting the develop branch. Paste the generated description into the PR body.\n6. **Report:** Output the URL of the newly created pull request.\n\n\nProceed with execution. User approval is not required for this standardized workflow.\n\n\n### 2.3 Managing Agent State and Failure Recovery\n\nEven with a robust architecture and well-crafted prompts, autonomous agents can fail. They can get stuck in loops, misinterpret instructions, or encounter unexpected environmental issues. A production-grade workflow must therefore include clear procedures for real-time course correction and failure recovery.\n\n#### Steering and Interruption\nCursor version 1.4 introduced crucial controls for real-time agent management.[3] These are the primary tools for manual intervention when an agent begins to drift:\n*

Queue Message (⌥+Enter / Alt+Enter): This allows the user to send a corrective instruction without immediately stopping the agent's current action. The message is queued and executed at the next logical break, typically after a tool call. This is useful for gentle course correction, e.g., "After you finish writing that file, please remember to also update the documentation."\n* **Immediate Interrupt (⌘+Enter / Ctrl+Enter):** This forcefully stops the agent's current generation and immediately processes the new instruction. This is the "emergency stop" button, used when the agent is clearly on the wrong path, e.g., "STOP. You are editing the wrong file. The correct file is authService.ts."\n\n#### Terminal Stability and Recovery\nThe reliability of agentic workflows that involve shell commands was significantly improved in version 1.6.[3] However, commands can still hang or fail. The agent must be prompted to handle these failures gracefully. A rule can be added: "When you execute a terminal command, you must check its exit code. If the exit code is non-zero, you must halt execution, report the failure and the command's output, and await further instructions." This prevents the agent from continuing with a flawed plan based on a failed prerequisite step.[2]\n\n#### Context Flushing: The Soft Reset\nAs previously discussed, the most common failure mode is contextual drift, where the agent becomes "confused" due to a cluttered or contradictory conversational history.[10] When steering and interruption fail to correct the

agent's behavior, the most effective recovery mechanism is a "controlled context reset." This involves:\n1. Stopping the current agent run.\n2. Starting a new chat session.\n3. Re-establishing the high-level strategic context by prompting the agent to read the

@llm-context.md file.\n4. Providing a new, more precise prompt for the original task, incorporating any lessons learned from the previous failure.\n\nThis pattern effectively flushes the noisy, short-term memory while retaining the stable, high-value context from the codebase index and project constitution. It is the go-to procedure for recovering from a severely derailed agent state.\n\n## III. Assurance and Safety: Validation, Review, and Security\n\nAn agent that generates code without a rigorous validation and safety framework is a liability, not an asset. This section details the critical processes that ensure the agent's output is correct, secure, and aligned with project standards. It covers the implementation of automated verification loops, the indispensable role of human-in-the-loop review, and a comprehensive protocol for hardening the agent against security threats. A production-grade agentic system requires a "Proof of Trust" model, where autonomy is not blindly granted but is constrained within a secure, sandboxed environment with auditable actions and strict, human-approved rules of engagement.[27] The default autonomous modes are insufficient for enterprise use without these additional layers of assurance.\n\n### 3.1 Implementing Automated Test and Verification Loops\n\nThe most powerful feature of an in-IDE agent is its ability to execute commands, allowing it to take responsibility for verifying the quality of its own output. This creates a tight feedback loop of

code→test→fix that can be fully automated.\n\n#### Configuring "YOLO Mode" Safely\nCursor's "YOLO Mode" allows the agent to automatically run terminal commands without explicit user approval for each one.[2] While the name implies recklessness, it can be configured to create a safe and powerful automation sandbox. Enabling this mode is a prerequisite for a self-verifying agent, but it must be done with carefully crafted guardrails.\n\n#### Crafting Allow/Deny Lists\nThe key to safe automation is defining a narrow, explicit command surface for the agent. In the YOLO Mode settings, a prompt should be used to establish a strict

allowlist and denylist.[25]\n\n

Allowlist Prompt Example:\n"You are permitted to run the following commands without seeking approval:\n* **Testing:** npm test, npm run test:unit, vitest, pytest\n* **Linting & Type Checking:** npm run lint, eslint --fix, prettier --write, tsc --noEmit\n* **Build Verification:** npm run build\n* **Filesystem (Read-only):** ls, cat, grep\n* **Filesystem (Write - limited):** touch, mkdir\n\nAny command not on this list requires explicit user confirmation."\n\n**Denylist Prompt Example:**\n"You are explicitly forbidden from running the following commands under any circumstances:\n* **Destructive Filesystem:** rm, mv\n* **Forceful Git Operations:** git push --force, git reset --hard\n* **System-level Changes:** sudo, apt-get, brew install\n\nViolation of these rules will result in immediate termination of the session."\n\nThese lists, combined with the agent's inherent inability to approve its own confirmation prompts for dangerous

commands, create a secure execution environment.\n\n#### The Test-Fix-Repeat Feedback Loop\nWith the safe sandbox configured, the core automated quality assurance workflow can be implemented. This loop is the heart of the test phase in the plan→code→test→review cycle.\n\nA playbook for this loop would instruct the agent:\n1. "After generating or modifying code, you will immediately run the relevant quality gate command (e.g., npm run lint for frontend changes, pytest for backend changes)."\n2. "You will capture the stdout and stderr from the command execution."\n3. "If the command exits with a code of 0 (success), you may proceed to the next step in your plan."\n4. "If the command exits with a non-zero code (failure), you will autonomously diagnose the errors from the output, generate a code change to fix them, and then re-run the command."\n5. "You will repeat this test-fix cycle until the command succeeds." [25]\n\nThis self-correcting loop ensures that the code presented for human review has already passed all automated quality checks, dramatically reducing the burden on the human reviewer and improving the overall quality of committed code.\n\n### 3.2 Human-in-the-Loop: The Review and Approval Workflow\n\nAutomation does not eliminate the need for human oversight; it elevates its importance. The human reviewer is the final and most critical quality gate, responsible for assessing the agent's work for architectural soundness, logical correctness, and alignment with business requirements—qualities that automated tests cannot fully capture. Cursor provides tools specifically designed to facilitate this crucial human-in-the-loop process.\n\n#### Leveraging the Diff & Review UI\nWhen an agent completes a task, it presents all proposed changes in a dedicated review interface.[22] This UI uses a familiar diff format, with color-coded lines indicating additions (green) and deletions (red). This is the primary control surface for the human reviewer.\n\nBest practices for this review stage include:\n*

Reviewing the Plan First: Before even looking at the code, review the agent's final report against its initial, approved plan. Did it accomplish all the steps? Did it deviate from the plan? Any deviations are a red flag that requires deeper investigation.\n* **Holistic File Review:** Use the file-by-file navigation to assess the changes in context. Do not just review the diffs in isolation. Open the full file to understand how the changes fit into the larger module.\n* **Selective Acceptance and Rejection:** The ability to accept or reject changes on a line-by-line or file-by-file basis provides the fine-grained control necessary for a thorough review.[22] A common workflow is to reject specific unwanted changes (e.g., unnecessary comments, debug

console.log statements) and then use the "Accept All" button for the rest.\n\n#### Providing Corrective Feedback\nIf a significant portion of the agent's work is rejected, it's crucial to provide corrective feedback. This closes the feedback loop and helps the agent improve. The process is:\n1. Reject the incorrect changes in the review UI.\n2. Start a follow-up prompt in the chat, referencing the rejected changes: "I have rejected your changes to userService.ts. The logic for calculating user permissions was incorrect. It failed to account for the 'auditor' role. Please review the requirements in @docs/permissions-spec.md and provide a corrected implementation."\n3. This initiates a new plan→code→test→review cycle focused on the specific correction.\n\n#### Agent Hooks for Auditing and Control\nThe Agent Hooks

feature, introduced in beta in version 1.7, provides a powerful new mechanism for programmatic oversight and control.[3] Hooks are custom scripts that can observe, influence, and even block actions within the agent's execution loop. They are the key to implementing enterprise-grade auditing and policy enforcement.\n\nA sample hook script (

.cursor/hooks/audit.js) could be written to create a comprehensive audit trail:\njavascript\n// Example audit hook concept\nexport function onBeforeRunCommand(command) {\n const logMessage = ` AGENT_COMMAND: ${command.tool} with args: ${JSON.stringify(command.args)}`;\n // Append logMessage to a local audit.log file\n console.log(logMessage);\n}\n\nexport function onAfterApplyEdit(edit) {\n const logMessage = ` AGENT_EDIT: Applied changes to ${edit.filePath}`;\n // Append logMessage to a local audit.log file\n console.log(logMessage);\n}\n\nThis ensures that a complete, immutable record of the agent's actions is maintained, which is essential for security audits and incident response.\n\n### 3.3 Agent Security Hardening\n\nAn autonomous agent with filesystem access and command execution capabilities must be treated as part of the application's attack surface. Hardening the agent's operating environment is not optional; it is a requirement for production deployment. The security protocol must address risks from data leakage, malicious instruction, and supply chain attacks.[30]\n\n#### Secrets and Sensitive Data Handling\nThe agent must be prevented from accessing or exfiltrating sensitive data.\n*

Context Sanitization: The .cursorignore file, which functions like .gitignore, must be configured to exclude all sensitive files and directories from the codebase index. This includes .env files, *.pem keys, and terraform state files.[17]\n*

Secret Scanning: A pre-commit hook using a tool like Gitleaks or Secretlint should be integrated into the repository.[31] The agent's core rules must instruct it to run and pass this hook before committing code. This provides an automated check against accidental secret exposure.\n*

Runtime Secrets: When the agent needs to use a secret (e.g., an API key to interact with an MCP tool), it should be instructed to read it from an environment variable. Cursor's support for interpolated variables in MCP configurations facilitates this secure pattern.[3]\n\n#### Prompt Injection and Malicious Rules\nAn attacker could potentially craft a malicious prompt or commit a compromised

.cursor/rules file to trick the agent into executing destructive commands or introducing vulnerabilities.[30]\n*

Rule File Integrity: All .cursor/rules and .cursor/commands files must be stored in version control. Changes to these files must be subject to a mandatory peer review process, just like application code. This ensures that the agent's core instructions are vetted.\n* **Input Sanitization:** While challenging, Agent Hooks can be used to implement basic input sanitization on prompts, blocking known malicious patterns or commands before they reach

the LLM.\n\n#### Sandboxed Execution\nThe agent's execution environment must be isolated from the host system to contain any potential malicious activity. This is the most robust defense against unforeseen agent behavior.\n* **Terminal Sandboxing:** The terminal used by the agent should be a containerized environment (e.g., Docker) rather than the host's native shell. This limits the agent's filesystem access to the project directory and prevents it from accessing or modifying system-level files.[32]\n*

Code Execution Sandboxing: For tasks that involve running newly generated code (e.g., in a Jupyter notebook context), a more secure sandbox like a WebAssembly runtime (Pyodide) or a micro-VM (Firecracker) should be used. These technologies provide strong isolation with minimal performance overhead.[33]\n\n#### Dependency Vetting\nAn agent instructed to add a new feature might autonomously decide to install a new dependency, creating a potential supply chain risk.\n*

Automated Vetting: The agent's rules must include a step to vet any new dependency before installation. The playbook should be: "Before adding a new npm package, you must first run npm audit on that package. If it reports any high or critical severity vulnerabilities, you must halt and report your findings for human review." [30]\n\nThis multi-layered security approach—combining context sanitization, instruction integrity, sandboxed execution, and automated vetting—creates a defensible architecture that allows for the safe operation of powerful autonomous agents in a production environment.\n\n## IV. Operational Excellence: Model Selection and Performance Measurement\n\nDeploying a coding agent is not a one-time setup; it is an ongoing process of optimization and measurement. Operational excellence requires making data-driven decisions about which Large Language Models (LLMs) to use for specific tasks and implementing a rigorous framework for quantifying the agent's impact on engineering productivity and code quality. Traditional developer metrics are often insufficient for capturing the value of agentic workflows, necessitating a new set of Key Performance Indicators (KPIs) tailored to this new paradigm.[35]\n\n### 4.1 The 2025 LLM Model Matrix for Code Generation\n\nSelecting the right LLM is a multi-faceted decision that involves balancing raw coding capability, speed, cost, and specific task suitability. A single "best" model does not exist; rather, there is an optimal model for a given task. The following matrix provides a data-driven framework for making this selection, synthesizing quantitative benchmarks, operational metrics, and qualitative community analysis from the June-September 2025 period.\n\n#### LLM Model Selection Matrix\n\n| Model Name | Provider | Version | SWE-bench Score (%) | Avg. Latency (ms) | Cost/1M Tokens (In/Out) | Context Window | Key Strengths | Key Weaknesses | Source(s) |\n| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |\n| GPT-5 | OpenAI |

gpt-5 | (Not publicly on SWE-bench) | High | High | 256k | Complex reasoning, architectural planning, high instruction adherence | 'Laziness', ignores queued instructions, can be verbose | [18] |\n| GPT-5 nano | OpenAI |

gpt-5-nano | 34.80 | Medium | Medium | 128k | Good balance of speed and intelligence for

general tasks | Less capable on highly complex problems than full GPT-5 | [37] |\n| Claude 3.7 Sonnet | Anthropic |

claude-3.7-sonnet | (Not publicly on SWE-bench) | Low | Low | 200k | High stability, speed, excellent for code generation and refactoring | Weaker on initial architectural design compared to GPT-5 | [39] |\n| Claude 4.5 Sonnet | Anthropic |

claude-4.5-sonnet | (Not publicly on SWE-bench) | Medium | Medium | 200k | Very stable, good for debugging, 'least lazy' model | Can be overly verbose, more expensive than 3.7 | [37] |\n| Gemini 2.5 Pro | Google |

gemini-2.5-pro | (Not publicly on SWE-bench) | Medium | Medium | 1M | Strong code generation, good for debugging | Prone to occasional execution breaks, less stable than Claude | [31] |\n| Deepseek Coder V3 | Deepseek |

deepseek-v3 | (Top performer on benchmarks) | Variable | Low (BYOK) | 128k | Excellent for pure code generation, open model flexibility | Requires self-hosting or third-party provider, less reasoning | [41] |\n\n#### Task-to-Model Mapping Strategy\nA sophisticated agentic workflow can leverage multiple models to optimize for cost and performance. This involves using different models for distinct phases of the

plan→code→test→review cycle:\n* **Planning Phase:** For tasks requiring deep reasoning, architectural design, or understanding complex user requirements, a high-intelligence model like **GPT-5** is the optimal choice. Its superior reasoning capabilities justify the higher cost and latency for this critical initial phase.\n* **Code Generation Phase:** Once a detailed plan is approved, the task shifts to implementation. For this phase, a faster and more cost-effective model optimized for code generation, such as **Claude 3.7 Sonnet**, is ideal. Its stability and speed make it well-suited for translating a detailed plan into correct and idiomatic code.\n* **Debugging and Iteration Phase:** When the agent is in a test-fix-repeat loop, a model with strong debugging capabilities and low latency is preferred. **Gemini 2.5 Pro** or **Claude 3.7 Sonnet** are both strong candidates, allowing for rapid iteration as the agent diagnoses and resolves errors.\n\nBy dynamically switching models based on the task at hand, teams can achieve better performance at a lower overall cost than relying on a single model for all operations.\n\n### 4.2 Key Performance Indicators (KPIs) for Agentic Workflows\n\nMeasuring the return on investment (ROI) of AI coding agents requires moving beyond traditional metrics like lines of code or commit frequency, which are poor proxies for value creation and can be easily gamed by an AI.[35] A new framework of KPIs is needed to measure the true impact on efficiency, quality, and developer experience.\n\n#### KPI Specification and Measurement\n\n| KPI Name | Description | Formula | Data Source(s) | Measurement Frequency | Target/Threshold |\n| :--- | :--- | :--- | :--- | :--- | :--- |\n|

Efficiency Metrics | | | | | |\n| Automated Cycle Time | Median time from task initiation (e.g., Linear issue creation) to the final PR merge for tasks completed by the agent. |

Median(PR_Merged_Timestamp - Task_Created_Timestamp) | Git Logs, Project Management API (Linear/Jira) | Per Sprint | >20% reduction vs. human baseline |\n| Human Intervention Rate | Average number of manual steering messages, interruptions, or rejected diffs per completed agent task. | Total_Interventions / Total_Agent_Tasks | IDE Telemetry, Chat Logs | Per Sprint | < 1.5 interventions/task |\n| **Quality & Reliability Metrics** | | | | |\n| Prompt→Commit Success Rate | Percentage of agent-generated code blocks that are accepted and committed by a human reviewer without significant manual rewrites. | (Accepted_AI_Suggestions / Total_AI_Suggestions) * 100 | IDE Telemetry, Git Logs | Per Sprint | > 85% |\n| AI Revert Percentage | Percentage of commits authored by the agent that are subsequently reverted via git revert. A direct measure of critical failures. | (Reverted_AI_Commits / Total_AI_Commits) * 100 | Git Logs | Monthly | < 2% |\n| Agent-Introduced Defect Density | Number of post-deployment bugs per 1,000 lines of code that are attributed to agent-authored commits. | (Bugs_From_AI_Code / AI_LOC_in_K) * 1000 | Git Blame, Issue Tracker | Quarterly | < project's human baseline |\n| **Adoption & Satisfaction Metrics** | | | | |\n| % Daily Agent Users | Percentage of active developers who invoke the agent at least once per day. Measures adoption and integration into daily workflows. | (Daily_Unique_Agent_Users / Total_Active_Developers) * 100 | IDE Telemetry | Weekly | > 60% of team |\n| Developer Satisfaction (NPS-style) | Qualitative feedback gathered via regular, short surveys. | Survey Question: "On a scale of 0-10, how likely are you to recommend the AI agent workflow to another developer?" | Internal Surveys | Monthly | Promoter Score > 40 |\n\n#### Implementation and Interpretation\nTo establish this measurement foundation, teams should run a pilot program. First, establish a baseline by measuring these metrics (where possible) for a two-week period before introducing the agent. Then, enable telemetry and begin tracking the agent-specific KPIs. Git logs can be parsed to identify agent-authored commits (e.g., by using a specific co-author tag like Co-Authored-By: cursor-agent <agent@org.com>).[35]\n\nEarly technical signals, such as Automated Cycle Time and Human Intervention Rate, should show improvement within one or two sprints. Quality metrics like AI Revert Percentage and Defect Density are lagging indicators and should be monitored quarterly. A drop in the Prompt→Commit Success Rate is a critical early warning sign that prompts, rules, or the selected model may be misaligned with the codebase, requiring immediate investigation.\n\nBy focusing on this balanced set of indicators—spanning efficiency, quality, and developer experience—engineering leaders can move beyond anecdotal claims and build a robust, quantitative business case for the adoption and scaling of autonomous agentic development.\n\n# Conclusions and Recommendations\n\nThe period from June to September 2025 has been formative for the field of autonomous software engineering, with tools like Cursor maturing from advanced assistants into platforms capable of supporting production-grade agentic workflows. The analysis of recent developments leads to a set of core conclusions and actionable recommendations for organizations seeking to deploy these capabilities reliably and safely.\n\n

1. Agent Governance Must Be Engineered, Not Prompted: The most significant finding is that reliable agent behavior is not an emergent property of a powerful LLM or a clever prompt. It is the result of a disciplined engineering approach that treats the agent's configuration as

code. The hierarchical system of Team Rules, Repository Rules, and Slash Commands provides the necessary declarative control to govern agent actions at scale. Organizations must invest in building and maintaining these rule artifacts with the same rigor they apply to their application code, including version control, code review, and modular design.\n\n**2. A Hybrid RAG Strategy is Essential for Mitigating Contextual Drift:** The primary failure mode of coding agents is contextual drift, where the agent loses sight of the project's strategic goals. The combination of automated, tactical context retrieval (e.g., @Codebase) and a manually curated, strategic context artifact (the llm-context.md file) creates a powerful hybrid RAG system. This approach anchors the agent's reasoning, significantly reducing the likelihood of it pursuing irrelevant or incorrect implementation paths. The llm-context.md should be considered a mandatory artifact for any serious agentic development project.\n\n**3. Autonomy Must Be Constrained by a "Proof of Trust" Security Model:** Granting an AI agent filesystem and shell access introduces significant security risks. A production deployment cannot operate on blind trust. It must implement a "Proof of Trust" model that combines declarative safety policies (allow/deny lists), runtime auditing and control (Agent Hooks), and environmental isolation (sandboxed execution). The principle of least privilege must be strictly applied, and the agent's autonomy should be confined within a secure, auditable sandbox.\n\n**4. Measurement Must Shift from Activity to Quality and Efficiency:** Traditional developer productivity metrics are obsolete in the context of AI-assisted development. A new set of KPIs, focused on outcomes rather than activity, is required to measure the true impact of agentic systems. Metrics such as **Prompt→Commit Success Rate**, **AI Revert Percentage**, and **Automated Cycle Time** provide a far more accurate picture of the agent's value and reliability. Organizations must adopt this new measurement framework to make informed decisions about tool adoption, model selection, and process optimization.\n\n**Recommendation for Implementation:**\n\nOrganizations should adopt a phased approach to deploying a production-grade Cursor agent:\n* **Phase 1: Foundation (1-2 Sprints):** Establish the foundational architecture. Define a comprehensive set of Team Rules for security and compliance. Develop repository-level rule files (.cursor/rules/) and a strategic llm-context.md for a single pilot project. Configure the security hardening checklist, including .cursorignore and secrets scanning.\n* **Phase 2: Workflow Development (2-3 Sprints):** With the foundation in place, develop a library of playbooks and Slash Commands for high-value tasks like TDD, bug fixing, and PR creation. Implement the automated test-fix-repeat loop by safely configuring YOLO mode with strict allow/deny lists. Train the pilot team on the human-in-the-loop review process and failure recovery procedures.\n* **Phase 3: Measurement and Optimization (Ongoing):** Implement the KPI framework to begin collecting baseline and ongoing performance data. Use this data to optimize the system by experimenting with different models for different tasks (as per the Model Matrix), refining prompts, and iterating on rule files. Based on successful KPI outcomes, scale the deployment to additional teams.\n\nBy following this structured, engineering-led approach, organizations can harness the transformative potential of autonomous coding agents while mitigating the inherent risks, ultimately building a faster, more efficient, and more reliable software development lifecycle.",

"playbooks":,
"rules":,
"prompts":,
"modelMatrix": {
"description": "A data-driven framework for selecting the optimal LLM for different coding tasks, balancing performance, cost, and qualitative factors. Data is synthesized from benchmarks and community reports from June-September 2025.",
"models":
},
{
"modelName": "GPT-5 nano",
"provider": "OpenAI",
"version": "gpt-5-nano",
"sweBenchScore": 34.8,
"avgLatencyMs": "Medium",
"costPerMillionTokens": {
"input": "Medium",
"output": "Medium"
},
"contextWindow": 128000,
"keyStrengths": "Good balance of speed and intelligence for general tasks",
"keyWeaknesses": "Less capable on highly complex problems than full GPT-5",
"sources": 37

},
{
"modelName": "Claude 3.7 Sonnet",
"provider": "Anthropic",
"version": "claude-3.7-sonnet",
"sweBenchScore": null,
"avgLatencyMs": "Low",
"costPerMillionTokens": {
"input": "Low",
"output": "Low"
},
"contextWindow": 200000,
"keyStrengths": "High stability, speed, excellent for code generation and refactoring",
"keyWeaknesses": "Weaker on initial architectural design compared to GPT-5",
"sources": 39

},
{
"modelName": "Claude 4.5 Sonnet",

"provider": "Anthropic",
"version": "claude-4.5-sonnet",
"sweBenchScore": null,
"avgLatencyMs": "Medium",
"costPerMillionTokens": {
"input": "Medium",
"output": "Medium"
},
"contextWindow": 200000,
"keyStrengths": "Very stable, good for debugging, 'least lazy' model",
"keyWeaknesses": "Can be overly verbose, more expensive than 3.7",
"sources": 37

},
{
"modelName": "Gemini 2.5 Pro",
"provider": "Google",
"version": "gemini-2.5-pro",
"sweBenchScore": null,
"avgLatencyMs": "Medium",
"costPerMillionTokens": {
"input": "Medium",
"output": "Medium"
},
"contextWindow": 1000000,
"keyStrengths": "Strong code generation, good for debugging",
"keyWeaknesses": "Prone to occasional execution breaks, less stable than Claude",
"sources": 31

},
{
"modelName": "Deepseek Coder V3",
"provider": "Deepseek",
"version": "deepseek-v3",
"sweBenchScore": "Top Performer",
"avgLatencyMs": "Variable",
"costPerMillionTokens": {
"input": "Low (BYOK)",
"output": "Low (BYOK)"
},
"contextWindow": 128000,
"keyStrengths": "Excellent for pure code generation, open model flexibility",
"keyWeaknesses": "Requires self-hosting or third-party provider, less reasoning capability",

"sources": 41

}
]
},
"kpiSpecs": {
"description": "A framework of Key Performance Indicators (KPIs) for measuring the effectiveness of AI coding agents, moving beyond traditional developer metrics to focus on efficiency, quality, and adoption.",
"kpis":
},
"securityChecklists":
},
{
"category": "Context Sanitization",
"control": "Enable Privacy Mode in settings.",
"implementation": "Navigate to Cursor Settings -> Privacy Mode and enable it.",
"riskMitigated": "Ensures session data and code are not used for model training by third parties.",
"sources": 31

},
{
"category": "Command Execution",
"control": "Configure strict command allow/deny lists for YOLO Mode.",
"implementation": "In YOLO Mode settings, provide a prompt that explicitly lists permitted commands (linters, testers, safe read-only commands) and denies destructive commands (rm, git push --force).",
"riskMitigated": "Prevents the agent from performing unauthorized or destructive actions on the filesystem or in version control.",
"sources": 25

},
{
"category": "Command Execution",
"control": "Implement sandboxed execution for the agent's terminal.",
"implementation": "Configure the agent's default terminal to be a containerized environment (e.g., Docker) with a volume mount limited to the project directory.",
"riskMitigated": "Isolates the agent from the host system, containing potential damage from malicious or buggy command execution.",
"sources": 32

},

{
"category": "Instruction Integrity",
"control": "Enforce mandatory peer review for all rule files.",
"implementation": "Store all .cursor/rules/ and .cursor/commands/ files in version control and protect the directory with branch protection rules requiring at least one approval for changes.",
"riskMitigated": "Prevents prompt injection and malicious instruction attacks via compromised rule files.",
"sources": 30

},
{
"category": "Instruction Integrity",
"control": "Implement an audit trail hook.",
"implementation": "Create a custom Agent Hook script (.cursor/hooks/audit.js) that logs all onBeforeRunCommand and onAfterApplyEdit events to an external, immutable log file.",
"riskMitigated": "Provides full transparency and an auditable record of all agent actions for security and incident response.",
"sources": 3

},
{
"category": "Supply Chain Security",
"control": "Implement automated secrets scanning.",
"implementation": "Set up a pre-commit hook using a tool like Gitleaks and instruct the agent in its rules that this hook must pass before it can commit.",
"riskMitigated": "Provides a final automated check against the agent accidentally committing secrets.",
"sources": 31

},
{
"category": "Supply Chain Security",
"control": "Enforce automated dependency vetting.",
"implementation": "Add a rule to the agent's doctrine requiring it to run npm audit or a similar tool on any new dependency and to halt for human approval if high/critical vulnerabilities are found.",
"riskMitigated": "Reduces the risk of the agent introducing vulnerable or malicious packages into the codebase.",
"sources": 30

}
]

}
],
"references": 1

}

Cytowane prace

1. My Cursor AI Workflow That Actually Works in Production | N's Blog - Namanyay Goel, otwierano: września 30, 2025, https://nmn.gl/blog/cursor-guide
2. cursor changelog | Cursor docs-Cursor Documentation-Cursor ai documentation - Cursor中文文档, otwierano: września 30, 2025, https://cursordocs.com/en/changelog
3. Changelog - Cursor, otwierano: września 30, 2025, https://cursor.com/changelog
4. chand1012/cursorrules: My personal LLM rules and how I make them - GitHub, otwierano: września 30, 2025, https://github.com/chand1012/cursorrules
5. cursor/cursor: The AI Code Editor - GitHub, otwierano: września 30, 2025, https://github.com/cursor/cursor
6. PatrickJS/awesome-cursorrules: Configuration files that ... - GitHub, otwierano: września 30, 2025, https://github.com/PatrickJS/awesome-cursorrules
7. GitHub Actions | Cursor Docs, otwierano: września 30, 2025, https://cursor.com/docs/cli/github-actions
8. MCP Directory | Cursor Docs, otwierano: września 30, 2025, https://cursor.com/docs/context/mcp/directory
9. Cursor AI: The AI-powered code editor changing the game - Daily.dev, otwierano: września 30, 2025, https://daily.dev/blog/cursor-ai-everything-you-should-know-about-the-new-ai-code-editor-in-one-place
10. I'm really disappointed with Cursor AI (Paid sub) - Reddit, otwierano: września 30, 2025, https://www.reddit.com/r/cursor/comments/1fk1tzg/im_really_disappointed_with_cursor_ai_paid_sub/
11. Cursor docs-Cursor Documentation-Cursor ai documentation - Cursor中文文档, otwierano: września 30, 2025, https://cursordocs.com/en
12. What is Cursor AI? Everything You Need to Know | UI Bakery Blog, otwierano: września 30, 2025, https://uibakery.io/blog/what-is-cursor-a
13. Cursor AI: A Guide With 10 Practical Examples - DataCamp, otwierano: września 30, 2025, https://www.datacamp.com/tutorial/cursor-ai-code-editor
14. Productive LLM Coding with an llm-context.md File - Donn Felker, otwierano: września 30, 2025, https://www.donnfelker.com/productive-llm-coding-with-an-llm-context-md-file/
15. TRAE IS GARBAGE!!!!!!! : r/Trae_ai - Reddit, otwierano: września 30, 2025, https://www.reddit.com/r/Trae_ai/comments/1n67twp/trae_is_garbage/
16. MCP Servers for Cursor - Cursor Directory, otwierano: września 30, 2025, https://cursor.directory/mcp

17. GitHub | Cursor Docs, otwierano: września 30, 2025, https://cursor.com/docs/integrations/github
18. Blog · Cursor, otwierano: września 30, 2025, https://cursor.com/blog
19. GPT-5 prompting guide | OpenAI Cookbook, otwierano: września 30, 2025, https://cookbook.openai.com/examples/gpt-5/gpt-5_prompting_guide
20. This gist provides structured prompting rules for optimizing Cursor AI interactions. It includes three key files to streamline AI behavior for different tasks. · GitHub, otwierano: września 30, 2025, https://gist.github.com/aashari/07cc9c1b6c0debbeb4f4d94a3a81339e
21. Cursor: The best way to code with AI, otwierano: września 30, 2025, https://cursor.com/
22. Diffs & Review | Cursor Docs, otwierano: września 30, 2025, https://cursor.com/docs/agent/review
23. Features · Cursor, otwierano: września 30, 2025, https://cursor.com/features
24. Composer Diff Viewer — Ultimate Cursor AI Course - Instructa.ai, otwierano: września 30, 2025, https://www.instructa.ai/course/cursor-ai/view/afcfdd73-c4d2-4b33-9e81-2cf2addb2522
25. How I use Cursor (+ my best tips) - Builder.io, otwierano: września 30, 2025, https://www.builder.io/blog/cursor-tips
26. Generate PR description from PR diff - Feature Requests - Cursor - Community Forum, otwierano: września 30, 2025, https://forum.cursor.com/t/generate-pr-description-from-pr-diff/20735
27. COTI-technical-whitepaper.pdf, otwierano: września 30, 2025, https://coti.io/files/COTI-technical-whitepaper.pdf
28. SNS Journal 2025, otwierano: września 30, 2025, https://smart-networks.europa.eu/wp-content/uploads/2025/05/sns-journal-2025-web-1.pdf
29. Commit Hooks are Critical with AI Agents in Cursor - Egghead.io, otwierano: września 30, 2025, https://egghead.io/commit-hooks-are-critical-with-ai-agents-in-cursor~jhoer
30. Cursor Security: Key Risks, Protections & Best Practices - Reco, otwierano: września 30, 2025, https://www.reco.ai/learn/cursor-security
31. The Developer's Cursor Checklist: Secure and Smart Practices for Using Cursor AI | TO THE NEW Blog, otwierano: września 30, 2025, https://www.tothenew.com/blog/the-developers-cursor-checklist-secure-and-smart-practices-for-using-cursor-ai/
32. Code Sandboxes for LLMs and AI Agents | Amir's Blog, otwierano: września 30, 2025, https://amirmalik.net/2025/03/07/code-sandboxes-for-llm-ai-agents
33. Sandboxing Agentic AI Workflows with WebAssembly | NVIDIA Technical Blog, otwierano: września 30, 2025, https://developer.nvidia.com/blog/sandboxing-agentic-ai-workflows-with-webassembly/
34. Self-host open-source LLM agent sandbox on your own cloud | SkyPilot Blog, otwierano: września 30, 2025, https://blog.skypilot.co/skypilot-llm-sandbox/

35. Autonomous Development Metrics: KPIs That Matter for AI-Assisted Engineering Teams, otwierano: września 30, 2025, https://www.augmentcode.com/guides/autonomous-development-metrics-kpis-that-matter-for-ai-assisted-engineering-teams
36. Measuring the productivity impact of AI coding tools: A practical guide for engineering leaders | Swarmia, otwierano: września 30, 2025, https://www.swarmia.com/blog/productivity-impact-of-ai-coding-tools/
37. How To Optimize Your Usage: The Best AI Models to Use, version 2.2 - Cursor Forum, otwierano: września 30, 2025, https://forum.cursor.com/t/how-to-optimize-your-usage-the-best-ai-models-to-use-version-2-2/131436
38. SWE-bench Leaderboards, otwierano: września 30, 2025, https://www.swebench.com/
39. oslook/cursor-ai-downloads: All Cursor AI's official download links for both the latest and older versions, making it easy for you to update, downgrade, and choose any version. - GitHub, otwierano: września 30, 2025, https://github.com/oslook/cursor-ai-downloads
40. 2025 AI Developer Tools Benchmark: Comprehensive IDE & Assistant Comparison, otwierano: września 30, 2025, https://kane.mx/posts/2025/ai-developer-tools-benchmark-comparison/
41. Best AI Coding Assistant 2025: Cline vs Cursor Comparison Guide, otwierano: września 30, 2025, https://cline.bot/blog/best-ai-coding-assistant-2025-complete-guide-to-cline-and-cursor
42. Cursor - Community Forum - The official forum to discuss Cursor., otwierano: września 30, 2025, https://forum.cursor.com/
43. Cursor - Documentation - Novita AI, otwierano: września 30, 2025, https://novita.ai/docs/guides/cursor
44. Measuring AI Developer Productivity Metrics That Actually Matter - Kinde, otwierano: września 30, 2025, https://kinde.com/learn/ai-for-software-engineering/managing-a-team/measuring-ai-developer-productivity-metrics-that-actually-matter/
45. Cursor AI: An In Depth Review in 2025 - Engine Labs Blog, otwierano: września 30, 2025, https://blog.enginelabs.ai/cursor-ai-an-in-depth-review
46. Cursor vs GitHub Copilot 2025: Same functionality but double the cost?, otwierano: września 30, 2025, https://community.latenode.com/t/cursor-vs-github-copilot-2025-same-functionality-but-double-the-cost/20812
47. Cursor vs Copilot: Which AI Coding Tool Is Better in 2025? - Openxcell, otwierano: września 30, 2025, https://www.openxcell.com/blog/cursor-vs-copilot/