ChatGPT

# Multi-agent coding rituals and file setup for a new computer

Software teams and research communities have begun adopting **multi-agent frameworks** to manage software development. Recent articles and projects (MetaGPT, ChatDev, AgileCoder, AgentMesh, etc.) show that **specialized LLM-based agents** produce higher-quality results than a single generalist agent by mimicking an Agile team. In these systems a **Planner** decomposes tasks, a **Coder/Developer** implements, a **Debugger/Tester** finds and fixes bugs, and a **Reviewer** or **QA** validates the result [1]. More advanced frameworks expand the team with roles such as **System Architect**, **Product Manager**, **UI/UX Designer**, **Technical Writer**, **DevOps Engineer**, **Security Engineer** and **Data Scientist** [2] [3]. The agents collaborate through a shared context and follow defined hand-off protocols; a central controller (sometimes called an **Admin agent**) orchestrates planning and ensures that each worker agent knows what others are doing [4]. Projects like **AgentMesh** demonstrate that decomposing work into Planner–Coder–Debugger–Reviewer improves performance by ensuring that each sub-task is handled by a specialist [5]. Similar ideas are appearing in the Cursor community—users request the ability to define **role-based agents via simple `.md` files** so that specialized agents can be invoked directly from chat [6].

To replicate these rituals on a new computer, you can create a folder structure with templates and configuration files that enforce a disciplined workflow. This guide summarises the roles, suggests file layouts and rule definitions, and provides command examples to set up a robust, multi-agent environment.

## 1. Choose your agent roles

Multi-agent systems mirror real engineering teams. The following roles and responsibilities are distilled from recent articles and community discussions:

| Role | Responsibilities (keywords only) | When critical |
|---|---|---|
| **Product Owner / Manager** | defines product requirements, prioritises features, balances user needs & business goals [2] | starting every project, approving PRDs |
| **System Architect / Software Architect** | strategic planning, defines architecture, coordinates tasks [7] | complex features, architectural decisions |
| **Software Engineer / Developer** | implements and tests code [7] | all development work |
| **QA Engineer / Tester / Reviewer** | designs test plans, finds bugs, ensures quality via regression testing [2] | all production code & critical systems |

| Role | Responsibilities (keywords only) | When critical |
|---|---|---|
| **UI/UX Designer** | crafts intuitive interfaces, conducts user research, creates wireframes/prototypes, ensures accessibility [3] | user-facing features, early design phases |
| **Technical Writer / Documentation Specialist** | writes API guides, tutorials, diagrams, maintains style guides [8] | after feature completion and architectural changes |
| **DevOps Engineer** | builds CI/CD pipelines, manages cloud & container infrastructure, automates deployment [9] | building & deploying production systems |
| **Security Engineer** | performs security assessments, penetration tests, implements authentication & authorization [10] | critical paths & before releases |
| **Data Scientist (optional)** | analyses data, builds ML models, creates dashboards [11] | projects with analytics or ML |

The **Admin agent** orchestrates the other agents; it breaks tasks into phases (plan/act) and manages the central context document [4]. You can emulate this by making yourself (the user) or your IDE the Product Owner who calls on other agents.

## 2. Core documentation files

Create a `docs/` directory with subfolders for each key document. These files provide the **long-term memory** and governance that multi-agent frameworks use to coordinate agents [4].

### 2.1 Product Requirements Document (`docs/prd/PRD.md`)

A **Product Requirement Document (PRD)** acts as the contract between the Product Owner and the rest of the team. Multi-agent frameworks emphasise that tasks should start with a high-level requirement which is refined and approved before coding [5]. Use a template with the following sections:

```
# <Project> — Product Requirements Document

## Metadata
- **Owner:** Your name
- **Status:** DISCOVERY \| DRAFT \| REVIEW \| FINAL
- **Last updated:** YYYY-MM-DD
- **Links:** Issue tracker, roadmap, or design documents

## Problem & Context
Describe the user problem, pain points, constraints and non-goals.

## Goals & Non-Goals
```

```
List SMART goals with success metrics; specify what is out of scope.

## Use Cases / User Stories
Write user stories or use-case bullets; include priorities and story points.

## Solution Overview
Sketch architecture ideas at a high level; link to Architecture.md for details.

## Acceptance Criteria
Enumerate testable statements; this will drive QA's test plan.

## Risks & Mitigations
Identify technical, product and compliance risks.

## Launch Plan
Define phases, roll-out strategy, telemetry requirements.

## QA Handoff
Leave space for developers to propose test plans and fixtures.

## Audit Trail
| Date (UTC) | Stage | Agent | Summary | Files | Commit |
|---|---|---|---|---|---|

## Review Notes
- APPROVED: Product Owner <date>
- APPROVED: QA <date>
```

Agents must update the **Audit Trail** at each stage to provide traceability. The Product Owner locks the PRD at the "Final" stage before development begins.

### 2.2 Architecture document (`docs/architecture/Architecture.md`)

The architecture file defines *how* the product will be built and acts as the **technical constitution**. Without clear architecture, multi-agent frameworks note that agents may produce conflicting solutions [4]. Include:

- **System Overview:** diagrams of components and responsibilities.
- **Data Flow:** input → processing → storage → output; note validation and error handling.
- **Governance & Guardrails:** data access rules, security controls, logging, compliance. This ensures agents respect data-governance boundaries.
- **Interfaces & Protocols:** API definitions, file formats, integration contracts.
- **Dependencies:** third-party services, versions, upgrade policy.
- **Non-Functional Requirements:** performance, scalability, reliability, security.
- **Change Process:** how to propose and approve architectural changes.

Agents should reference this document when generating code, and your rules should prevent coding if the architecture is missing or incomplete.

### 2.3 UX and UI design (`docs/ux/UX_UI.md`)

User interfaces need as much formalisation as backend architecture. A UI/UX Designer agent should document:

- **Design Principles:** accessibility, consistency, visual hierarchy, colour and typography guidelines [3].
- **User Journeys:** primary flows and edge cases.
- **Wireframes / Mock-ups:** links to Figma, or ASCII sketches if necessary.
- **Interaction Rules:** navigation hierarchy, error feedback, responsive breakpoints.
- **Constraints:** device assumptions, localisation, performance requirements.

### 2.4 Audit trail / long-term memory (`docs/audit/AuditTrail.md`)

If you prefer a standalone audit log instead of embedding it in the PRD, maintain a table of contributions (date, agent, description, files, commit). This helps reconstruct the sequence of decisions and is recommended by multi-agent frameworks to maintain institutional memory [4].

## 3. AGENTS.md: define your team

`AGENTS.md` is a Markdown "manifesto" listing each agent. Cursor docs emphasise that nested `AGENTS.md` files override parent ones and act as the primary way to define agent behaviours in the IDE [6]. A typical structure:

```
# AGENTS.md — Team Manifesto

## Global Working Agreement
- Follow the PRD pipeline and respect stage gates.
- Adhere to Architecture.md and UX_UI.md before implementing.
- Update Audit Trail with every meaningful change.
- Use commit trailers to indicate agent, stage and artifact.

## ProductOwner
**Role:** Provide vision, prioritise backlog, approve requirements.
**Must do:** Keep PRD current; add user stories; sign off on final PRD.

## SystemArchitect
**Role:** Plan technical strategy, design architecture, coordinate tasks.
**Must do:** Maintain Architecture.md; highlight risks; ensure decisions align
with architecture.

## SoftwareEngineer
**Role:** Write and test code.
**Must do:** Wait for PRD approval; follow architecture; create `QA Handoff`
section in PRD with a test plan; reference architecture and UX docs in code
comments.
```

```
## QAEngineer
**Role:** Validate functionality and quality.
**Must do:** Design comprehensive test plans; ensure acceptance criteria met;
write `APPROVED: QA` note when satisfied.

## UXDesigner
**Role:** Design user interfaces and interactions.
**Must do:** Maintain UX_UI.md with personas, wireframes, accessibility notes;
review PRD for UX implications.

## TechnicalWriter
**Role:** Produce and maintain documentation.
**Must do:** Create API docs and tutorials; keep docs up to date with code
changes.

## DevOpsEngineer
**Role:** Manage CI/CD pipelines and infrastructure.
**Must do:** Build deployment scripts; ensure environment consistency; document
infrastructure.

## SecurityEngineer
**Role:** Protect the system.
**Must do:** Run security audits; propose mitigations; approve releases.

## DataScientist (optional)
**Role:** Analyse data and build ML models.
**Must do:** Define data requirements; build analytics; share findings via
dashboards.
```

Agents should include **handoff protocols**. For example, the SoftwareEngineer adds a `QA Handoff` section with what to test; the QAEngineer waits until this section exists. These explicit cues replicate the plan/act protocol used by frameworks like AgentMesh [12].

## 4. Rules and pipelines in `.cursor/rules/*.mdc`

Cursor supports **rules** defined in `.mdc` files. These act as your system prompts and can enforce stage gates across all agents. A global rule might look like this (adapt and shorten as needed):

```
---  /* .cursor/rules/prd-pipeline.mdc */
description: "PRD pipeline and review gates"
globs:
  - docs/prd/**/*.md
alwaysApply: true
---
# PRD Pipeline (Enforced)
Stages:
```

```
  1) Discovery → output "CHECKPOINT: DISCOVERY" with bullets for problem, users,
constraints.
  2) Draft → output "CHECKPOINT: DRAFT" with full PRD skeleton filled.
  3) Review → Planner requests reviews from ProductOwner and QA.  Require both
to add "APPROVED: <role>" notes.
  4) Finalization → lock scope, add success metrics and ship plan.

Handoff Protocol:
  - Developer starts only after `APPROVED: ProductOwner`.
  - QA starts only after "QA Handoff" section exists with test plan.
  - Every stage must update the Audit Trail.

Commit Convention:
  - Commit titles: `feat(prd): [stage:<DISCOVERY|DRAFT|REVIEW|FINAL>] <summary>`
  - Commit trailers (required):
      Agent: <ProductOwner|SystemArchitect|SoftwareEngineer|QAEngineer|
UXDesigner|...>
      Stage: <DISCOVERY|DRAFT|REVIEW|FINAL>
      Artifact: docs/prd/<file>.md
```

Create additional rule files for architecture enforcement (e.g., block code generation if Architecture.md or UX_UI.md are missing) or security gates. Cursor's **Rules for AI** setting also allows personalising behaviour at the application level; you can specify coding standards, naming conventions, or functional programming preferences [13].

## 5. Commands: script your rituals

Cursor allows you to define reusable **commands** stored under `.cursor/commands`. These are Markdown files named after the command (e.g., `create-prd.md`). When you type `/` in the chat, commands appear automatically [14] [15]. A command file can standardise multi-step rituals. Examples:

- `create-prd.md` – Ask the ProductOwner to generate a new `PRD.md` from the template, set status to DISCOVERY, and add initial user stories.
- `plan-feature.md` – Instruct the SystemArchitect to break down a new feature into tasks, referencing Architecture.md and UX_UI.md.
- `implement-feature.md` – Tell the SoftwareEngineer to implement the planned tasks, update the `QA Handoff` section, and commit with proper trailers.
- `review-code.md` – Provide a checklist for QAEngineer and SecurityEngineer to review functionality, code quality, and security (modelled after the code review command examples in Cursor docs). It may include verifying correct functionality, checking for security vulnerabilities, and ensuring the code follows style and naming conventions [16].
- `security-audit.md` – Ask the SecurityEngineer to run automated security checks (e.g., OWASP Top 10) and update the Audit Trail. [17].
- `new-release.md` – Instruct the DevOpsEngineer to prepare deployment, run CI/CD pipelines, and produce release notes. The TechnicalWriter then finalises documentation.

Storing commands with descriptive names ensures that everyone (including AI agents) follows the same ritual when creating features, performing reviews, or shipping releases. Because commands are just Markdown, you can version and edit them like code.

## 6. Additional context and persistent notes

To maximise AI context, maintain a comprehensive `README.md` at the project root that explains the project's purpose, architecture, key technologies and conventions [18]. For complex subsystems or domain logic, create separate docs (e.g., `docs/security/ThreatModel.md`) and reference them from prompts. Cursor's **Notepads** feature (still in beta) can store persistent notes that agents can reference across sessions; use it for frequently needed context such as design tokens or API endpoints [19].

## 7. Setting up your new computer

1. **Install Cursor and enable Agent Mode.** Follow the official installation instructions and ensure that the IDE can run terminal commands. Consider enabling **YOLO mode** only in development environments, as it allows the agent to run commands without confirmation [20].
2. **Clone or create your repository** with the directory structure shown below and commit the template files. For example:

```
docs/
  prd/PRD.md                  # product requirements template
  architecture/Architecture.md
  ux/UX_UI.md
  audit/AuditTrail.md         # optional separate log
.cursor/
  rules/prd-pipeline.mdc      # enforce PRD process
  rules/architecture.mdc      # enforce architecture & UX docs
  commands/create-prd.md      # command definitions
  commands/plan-feature.md
  commands/implement-feature.md
  commands/review-code.md
  commands/security-audit.md
  commands/new-release.md
AGENTS.md                     # team manifesto
README.md                     # project overview
src/                          # your codebase
```

1. **Define global rules** in `.cursor/rules` and write your `AGENTS.md` based on Section 3. Tailor responsibilities and handoff protocols to your workflow.
2. **Write command files** under `.cursor/commands`. Start with at least `create-prd`, `plan-feature`, `implement-feature` and `review-code`. Use the examples from the Cursor docs as inspiration [16].
3. **Populate the PRD, Architecture, and UX docs** with real information. Without these, your rules should prevent agents from writing code.

4. **Use commit trailers** to record the agent, stage, and artifact in each commit. Optionally write a `commit-msg` Git hook to enforce this convention.
5. **Iterate and refine.** Multi-agent workflows often require tuning. As you discover new patterns, update your rules, commands and agent definitions.

## 8. Lessons from the community

- **Specialisation matters.** GitHub projects show that dividing work among System Architect, Security Reviewer, Context Specialist, Feature Developer and Documentation Writer improves quality and reduces context overload [7]. Multi-agent solutions emphasise that each agent should handle only tasks where it adds value [21].
- **Persistent context is essential.** The Main Context Document (architecture, API endpoints, data models, UI components) becomes the shared ground truth for all agents [4]. Without it, agents re-invent or contradict each other.
- **Plan/Act protocol prevents conflicts.** Agent MCP uses Plan Mode (query context, determine dependencies) and Act Mode (execute tasks, update metadata) to coordinate work [22]. You can mimic this by requiring planning (Planner agent produces a numbered plan) before any implementation.
- **Quality gates are non-negotiable.** Projects like AgentMesh include security and quality reviews as mandatory checkpoints [21]. Build similar gates into your rules so that every major change is validated by QA and Security before merging.
- **User-facing design is a first-class citizen.** Multi-agent frameworks include UI/UX Designers and Technical Writers. Provide them with dedicated documentation (`UX_UI.md`) and ensure they sign off on user-facing features [3] [8].
- **Automate repetitive rituals.** Commands simplify common workflows. Use them to bootstrap new PRDs, plan features, implement tasks, conduct reviews and security audits.
- **Keep security in mind.** The community asks for built-in automated security testing and specialized security agents [17]. Include security commands and rules (e.g., run OWASP checks) so that your system doesn't ship vulnerable code.

## Conclusion

The GenAI revolution has shown that coding assistants are no longer limited to autocomplete suggestions. By formalising your workflows through documentation, clearly defined roles, rules and commands, you can build a robust multi-agent system that mirrors a real Agile team. Investing time upfront in these templates and pipelines will ensure that your AI collaborators produce consistent, high-quality work and that your own role as Product Owner becomes one of orchestration rather than firefighting. Adopt the rituals described here to set your system "in stone" and enable continuous, disciplined development.

---

[1] [5] AgentMesh: A Cooperative Multi-Agent Generative AI Framework for Software Development Automation
https://arxiv.org/html/2507.19902v1

[2] [3] [8] [9] [10] [11] Reddit - The heart of the internet
https://www.reddit.com/r/RooCode/comments/1i8b25u/resource_7_specialized_ai_agents_for_complete/

4 22 Agent MCP: The Multi-Agent Framework That Changed How I Build Software : r/cursor
https://www.reddit.com/r/cursor/comments/1klrq64/agent_mcp_the_multiagent_framework_that_changed/

6 17 Multi-Agent Task Automation with Role-Based .md Configurations & Automated Security Testing for SaaS - Feature Requests - Cursor - Community Forum
https://forum.cursor.com/t/multi-agent-task-automation-with-role-based-md-configurations-automated-security-testing-for-saas/131026

7 21 cursor-agents/readme.md at master · pridiuksson/cursor-agents · GitHub
https://github.com/pridiuksson/cursor-agents/blob/master/readme.md

12 cursor-agents/process.md at master · pridiuksson/cursor-agents · GitHub
https://github.com/pridiuksson/cursor-agents/blob/master/process.md

13 18 19 20 How to Use Cursor Agent Mode
https://apidog.com/blog/how-to-use-cursor-agent-mode/

14 Commands | Cursor Docs
https://cursor.com/docs/agent/chat/commands%23creating-commands

15 Commands | Cursor Docs
https://cursor.com/docs/agent/chat/commands%23how-commands-work

16 Commands | Cursor Docs
https://cursor.com/docs/agent/chat/commands%23examples