

Multi-agent PRD automation with MetaGPT, Ollama and DeepSeek

Learn how to build an AI-powered tool with MetaGPT, DeepSeek and Ollama that helps product managers quickly create comprehensive product requirement documents (PRDs) by using a team of specialized AI agents.

MetaGPT is a multi-agent framework developed by DeepWisdom, a tech startup focused on developing open source tools that automate work by using artificial intelligence, multi-agent systems and agentic workflows.

Unlike a single-agent approach, where one model attempts to handle all aspects of the task, this multi-agent system assigns each agent a specific role and clearly defined responsibilities. By following structured workflows and reviewing each other's outputs, the team collectively generates a high-quality PRD that is more aligned with stakeholder goals, better organized and less prone to oversight.

Before we begin, here's a few terms to help familiarize yourself with the application's tech stack:

MetaGPT: A framework that structures large language model (LLM) agents into collaborative roles, enabling them to work together like a coordinated team.

Ollama: A local runtime for running and managing open source LLMs directly on your personal computer or workstation.

DeepSeek: An open source language model optimized for tasks like research, reasoning and technical writing.

Automating PRDs with multi-agent collaboration

Creating PRDs can be time-consuming but artificial intelligence can assist by accelerating the process of completion.

Multi-agent collaboration is implemented in frameworks like MetaGPT, an AI tool that orchestrates the coordination of multiple role-playing agents to complete a complex task. A complex task can be considered anything that requires more than one step to complete.

AI PRD creation is an excellent use case for multi-agent collaboration because it mirrors a real-world product development process, where multiple stakeholders contribute to stages such as research, planning, review and refinement. To get the full advantage of AI-generated content, one should consider using a multi-agent system versus a single chatbot like OpenAI's ChatGPT or Microsoft's Copilot.

Why use a multi-agent system like MetaGPT?

MetaGPT uses specialized AI agents with distinct roles where each different role can be customized to fit nearly any workflow with minimal coding. This flexibility is possible because of the LLM's strong ability to understand natural language. Users define agent behaviors and workflows through prompt engineering and lightweight software development.

The goal of MetaGPT is to enable effective multi-agent collaboration. By simulating a structured team, it enables role-specific reasoning and task delegation, producing more context-aware and consistent outputs like high-quality PRDs.

Later in this tutorial, we'll show how a single agent generates an initial PRD draft—similar to using a stand-alone chatbot. We'll then compare this draft to the final, more accurate PRD produced through multi-agent collaboration. This method will demonstrate how teamwork improves quality beyond what one agent can achieve alone.

Why use DeepSeek for PRD automation?

[DeepSeek](#), developed by DeepSeek-AI, is a family of cutting-edge open-source LLMs optimized for reasoning tasks, structured content creation and efficient AI development workflows. In this project, we use `deepseek-r1`, a performant base model ideal for automating product documentation.

Here's why DeepSeek stands out for building PRDs with a multi-agent system like MetaGPT:

- **Structured output for automation:** DeepSeek models generate consistent markdown output, which aligns well with workflows that require formal document structures such as PRDs or technical specs.
- **Reasoning capabilities:** The model supports multiagent interaction loops by handling sequential reasoning and revision steps.
- **Performance benchmarks:** According to published benchmarks, DeepSeek's models perform competitively with other open-source models in the 7–13B parameter range, including those from Mistral, LLaMA and IBM® Granite® models. Granite, developed by IBM Research®, is designed for enterprise-grade use cases, with a strong emphasis on governance, robustness and structured business reasoning.
- **Local inference:** Running `deepseek-r1` through Ollama on local GPUs enables lower-latency experimentation without relying on external application programming interfaces (APIs) like OpenAI or Microsoft Azure endpoints (no need for an API key!). This approach can be useful for workflows requiring data privacy or offline development.
- **Language support and context window:** DeepSeek offers strong multilingual support, including Chinese, and includes a reasonably long context window, which supports extended memory across multiagent sessions.

While DeepSeek is used in this tutorial, the same multi-agent system can be configured to run with other LLMs compatible with Ollama, Hugging Face or OpenAI's API. The choice of model depends on the tradeoff between reasoning accuracy, output structure, resource availability and intended deployment environment.

How does MetaGPT work?

MetaGPT uses the concept of standard operating procedures (SOPs) to align human and AI collaboration by structuring workflows based on real-world teams (that is, a software company or product development team).

A SOP provides detailed, step-by-step guidelines for completing a specific task or process. MetaGPT applies this concept by decomposing complex tasks (like creating a PRD) into clear, actionable steps.

Each action is assigned to a designated “team member” or role-playing AI agent.

MetaGPT base agent

MetaGPT agents operate within a structured, role-based system designed to simulate and coordinate their tasks through collaborative workflows.

Each agent follows an organized agentic workflow grounded in **four** core concepts:

1. **Role:** A specialized persona to achieve a specific purpose (project manager, designer, analyst).
2. **Action:** The ability to perform certain tasks (write, review, research and more).
3. **Memory:** Individual memory is stored as a list of message objects that include past interactions, observations and actions. These messages are published to a shared message pool for communication between agents. Memories inform agent actions.
4. **Environment:** A common space (the global message pool) to access information from other agents without direct interaction. This space acts as a shared context for all agents.

Together, these components form the foundation for agent autonomy and task execution in MetaGPT. Next, we'll explore how these agents communicate and collaborate to complete multistep tasks like generating a PRD.

How MetaGPT agents work together

MetaGPT agents follow a coordinated process where each agent contributes to a shared goal. Each agent processes information and reasons based on its role, takes action and shares results with others. This approach enables a dynamic, step-by-step collaboration that builds toward the final output

MetaGPT agent workflow:

1. **Observe:** The agent reviews the current states (for example, the latest PRD draft).
2. **Think:** Using the LLM, it decides what to do next based on its roles and the available information.
3. **Act:** The agent performs its assigned task—such as writing, reviewing or researching.
4. **Share:** The agent logs its output and broadcasts a message to the shared environment for other agents on the team to access.
5. **Next agent:** The process moves to the next agent who picks up where the last one left off and repeats the process until consensus is reached.

Agents iterate on this structured loop, building on each other's work in each round until reaching a final, more complete and accurate output.

With MetaGPT, it's possible to build a fully automated AI product development team by customizing agent roles, SOPs, PRD templates, stakeholder priorities and overall project goals. The framework is extensible, allowing teams to adapt it to specific workflows and requirements.

Now that we understand how individual agents operate and collaborate, let's look at how this process is orchestrated at the application level in the full PRD generation workflow.

How the multiagent PRD workflow operates

This section acts as a step-by-step guide to understand the workflow of this multiagent PRD generation application's team of MetaGPT agents.

Defining the standard operating procedure (SOP)

Let's define a structured agentic workflow with our MetaGPT team by creating a SOP. This SOP breaks down the complex task of creating a PRD into clear, actionable steps, assigning each to a specialized agent.

Roles and responsibilities

A well-defined SOP clarifies each agent's role and actions. This structure promotes accountability and smooth execution across the PRD lifecycle: drafting, research enrichment, peer review and revision.

Team roles:

- **Product manager (team lead):** Orchestrates the workflow, drafts the initial PRD, collects research and reviews feedback, revises the document and saves all versions. The project manager (PM) agent leads the process and coordinates the other agents.
- **Researcher:** Enriches the PRD with relevant research and supporting data.
- **Reviewer:** Reviews the PRD and provides actionable feedback for improvement.

Workflow stages

1. **User idea:** The user provides a project idea ("Write a PRD for a banking application for wealth management") through the command line.
2. **Team setup:** The app creates a team and assigns the roles: product manager, researcher, reviewer.
3. **Drafting:** The product manager (as team lead) generates and saves the initial PRD as DraftPRD.md that outlines the products goals, user personas, key features and functional requirements.
4. **Research:** The researcher reviews the draft and provides supporting research.
5. **Review:** The reviewer examines the draft and gives feedback.
6. **Revision:** The PM collects the research and review feedback, revises the PRD and saves the final document as PRD.md.
7. **Output:** The final PRD (with research and revisions) is saved as a markdown file in the project directory.

This SOP ensures that the project manager leads the team, coordinating all contributions to automate the creation of a research-backed and reviewed PRD.

System requirements

To run this tutorial effectively, users need the following requirements:

- **Operating system:** macOS, Linux or Windows
- **Memory (RAM):** ≥ 16 GB
- **Disk space:** ≥ 10 GB free (for Python environment, Ollama models and generated files)
- **Ollama:** Installed and running locally (default port 11434)
- **Python version:** 3.11.x

Note: Running larger models or multiple agents might require more memory (32 GB+ recommended for best performance). Intermittent timeout errors can occur. If you encounter timeout errors, try restarting the process and ensuring your system has sufficient resources.

Steps

Step 1. Create a venv

Step 2. Install MetaGPT

Install the latest development version of MetaGPT.

```
pip install git+https://github.com/geekan/MetaGPT
```

Powered by  Granite

Important: For this tutorial, you must install MetaGPT by using the command above. Do **not** install MetaGPT from PyPI or other sources, as only the latest development version is supported here.

Step 3. Install Ollama

Step 4. Start the Ollama server and pull deepseek-r1:8b

After installation, you can start the Ollama server and pull a model (deepseek-r1:8b) with:

```
ollama serve ollama pull deepseek-r1:8b
```

Powered by  Granite

Step 5. Configure MetaGPT to use Ollama

To configure the Ollama and Deepseek to work with MetaGPT, we need to create and edit a config file.

Initialize the MetaGPT configuration:

```
metagpt --init-config
```

Powered by  Granite

This action creates a file at `~/metagpt/config2.yaml`

Edit the file to configure your LLM with the following steps:


1. In a terminal window, run the following command to open the config file in the nano editor:

```
nano ~/metagpt/config2.yaml
```

Powered by  Granite

2. Edit the file to match this Ollama configuration that uses the deepseek-r1:8b model.

```
llm: api_type: 'ollama' base_url: 'http://127.0.0.1:11434/api' model: 'deepseek-r1:8b'
```

Powered by  Granite

Note: If the field `api_key` appears in the YAML file, do not leave it blank. Either provide a valid key or remove the field entirely. The program will not run if `api_key` exists and is empty.

1. After you've made the preceding changes, press `Ctrl + O` to save, then press `Enter` to confirm.
2. Press `Ctrl + X` to exit nano

Your LLM configuration changes are now saved!

For additional config examples, see the two provided in the MetaGPT docs [here](#) and [here](#).

Step 6. Learn how MetaGPT agents work: Actions and roles

MetaGPT agents are built from two main components:

- **Actions:** Discrete tasks or operations an agent can perform (for example, writing a PRD, conducting research).
- **Roles:** Defines the agent's responsibilities and which actions it can take (for example, project manager, researcher).

Actions

An Action is a Python class that defines a specific task for an agent.

Actions tell each agent what to do and how to interact with the language model.

Each action typically includes:

- `PROMPT_TEMPLATE` : The instruction or message sent to the LLM (for example, "Write a PRD in markdown format").
- `run()` method: Fills in the prompt template, sends it to the LLM and returns the model's response.
- Optionally, `parse_text()` method: Processes the LLM's output to extract the relevant information (such as markdown, code or JSON).

Require imports for actions:

```
import re
import os
from metagpt.actions import Action
```

Powered by  Granite

- `re` is for regular expressions (used in `parse_text`)
- `os` is for file operations (used in `save_prd`)
- `Action` is the base class for all actions in MetaGPT

Roles

The role class represents an AI agent or team member in the workflow. Roles instruct the model how to act and define which specific part of the process it should follow (such as managing, researching or reviewing).

Each role typically includes:

- `__init__` : Initializes the role, sets up its actions and defines what events or messages it should watch for.
- `__act` : Executes one or more assigned action when it's the agents turn to act. This method defines the agent's behavior in the workflow.

Required imports for roles:

```
from metagpt.roles import Role
from metagpt.schema import Message
from metagpt.logs import logger
```

Powered by  Granite

- Role is a base class for all agent roles in MetaGPT.
- Message is used to return results from actions.
- logger is used for logging output and debugging information.

Workflow overview

MetaGPT organizes the workflow into rounds, which are iterative cycles where agents collaborate to improve the PRD. Each round consists of the following steps:

Round 1: Initial draft

- The project manager creates and saves the first PRD draft based on the user's prompt.
- The researcher and reviewer receive this draft for their tasks.

Round 2 (and beyond): Review and revision:

- The researcher generates supporting research for the PRD.
- The reviewer provides feedback on the PRD draft.
- The project manager revises the PRD using the new research and review feedback, then saves the updated version.

Repeat

- The process can repeat for multiple rounds, allowing the PRD to be incrementally improved with each cycle.

Multiagent PRD generation workflow diagram:

```
User prompt → Team initialization → PRD draft (Project Manager) → Research and review (Researcher & Reviewer) → Draft revision (Project Manager) → Save final PRD
```

Powered by  Granite

In the next step, you'll build a team of agents for PRD AI automation. We'll define each agent's role and connect its relevant workflow actions.

Step 7. Build a multiagent PRD team with MetaGPT

In this section, you'll see how to define agent **actions**, create agent **roles** and assemble a team to automate PRD generation, research and review.

Define agent actions

Here are the agent actions that the PRD team will perform by using the Action class:

```
import re
import os
from metagpt.actions import Action

def clean_response(rsp):  # Cleans LLM output, extracting markdown and removing extra tags
    rsp = re.sub(r"<think>.*?</think>", "", rsp, flags=re.DOTALL)
    pattern = r"```(?:markdown)?(?:.*)```"
    match = re.search(pattern, rsp, re.DOTALL)
    text = match.group(1) if match else rsp
    return text.strip()


class WritePRD(Action):
    PROMPT_TEMPLATE: str = """
    Write a comprehensive product requirements document (PRD) for {instruction} and provide the output in markdown format.
    **Important:**
    - Do NOT include any code, programming language, or technical implementation details.
    - Only write markdown for a PRD document (sections like Introduction, Goals, User Stories, Requirements, etc.).
    - Do NOT include code blocks, scripts, or pseudocode.
    - Limit your response to a maximum of 1,500-3,000 words and no more than 7 unique sections.
    - Ensure that no sections are repeated.
    - Ensure that each section is ordered and formatted correctly with appropriate headings and subheadings.
    Return ``your markdown text here with NO other texts, your text: """
    name: str = "WritePRD"
    async def run(self, instruction: str):
        prompt = self.PROMPT_TEMPLATE.format(instruction=instruction)
        rsp = await self._aask(prompt)
        prd_text = self.parse_text(rsp)
        return prd_text
    @staticmethod
    def parse_text(rsp):
        return clean_response(rsp)

class SavePRD(Action):
    name: str = "SavePRD"
    async def run(self, content: str, filename: str = "PRD.md"):
        filepath = os.path.join(os.getcwd(), filename)
        with open(filepath, "w", encoding="utf-8") as f:
            f.write(content)
        return f"PRD saved to {filepath}"

class ConductResearch(Action):
    PROMPT_TEMPLATE: str = """
    Context: {context}
    You are a research assistant working with the Project Manager to ensure that the PRD includes information from a detailed research report for the given PRD.
    Use the {instruction} to generate a detailed research report on relevant details that should be included in the PRD and provide the output in markdown format.
    Include relevant data, statistics, and references to support the PRD.
    **Important:**
    1. Return only the markdown text.
    2. Do not include any other text or explanations.
    3. Limit your response to the content that is relevant to the PRD and a maximum of 500-1,500 words.
    Return ``your markdown text here`` with NO other texts, your text: """
    name: str = "ConductResearch"
    async def run(self, instruction: str, context: str = ""):
        prompt = self.PROMPT_TEMPLATE.format(instruction=instruction, context=context)
        rsp = await self._aask(prompt)
        research_content = self.parse_text(rsp)
        return research_content
    @staticmethod
    def parse_text(rsp):
        return clean_response(rsp)

class PerformReview(Action):
    PROMPT_TEMPLATE: str = """
    You are a product reviewer. The following is a Product Requirements Document (PRD) generated for a project. Please review the PRD below and provide critical, actionable feedback to improve its clarity, completeness, and effectiveness. Highlight any missing sections, unclear requirements, or potential risks. Ensure that no sections are repeated.
    **Important:**
    1. Return only the markdown text.
    2. Do not include any other text or explanations.
    3. Limit your response to the content that is relevant to the PRD.
    4. Limit your response to a maximum of 500-1,000 words.
    Return your feedback in markdown format only.
    PRD to review: {context}
    """
    name: str = "PerformReview"
    async def run(self, context: str):
        prompt = self.PROMPT_TEMPLATE.format(context=context)
        rsp = await self._aask(prompt)
        review_content = self.parse_text(rsp)
        return review_content
    @staticmethod
    def parse_text(rsp):
        return clean_response(rsp)

class RevisePRD(Action):
    PROMPT_TEMPLATE: str = """
    Revise the Product Requirements Document (PRD) based on the following review feedback. Revise the PRD to address all reviewer suggestions, clarifying vague terms, adding measurable goals, expanding on integrations, including user stories, functional requirements, and adding any missing sections as suggested.
    **Important:**
    1. Return only the markdown text.
    2. Do not include any other text or explanations.
    3. Include a section at the end titled "Document revision notes" that summarizes the key revisions.
    4. Limit your response to a maximum of 1,500-4,000 words and no more than unique 12 sections.
    5. Ensure that no sections are repeated.
    6. Ensure that each section is ordered and formatted correctly with appropriate headings and subheadings.
    PRD: {prd}
    Review Feedback: {review}
    Return ``your markdown text here`` with NO other texts, your text: """
    name: str = "RevisePRD"
    async def run(self, prd: str, review: str):
        prompt = self.PROMPT_TEMPLATE.format(prd=prd, review=review)
        rsp =
```

```
await self._ask(prompt)    revised_prd = self.parse_text(rsp)    return revised_prd
@staticmethod    def parse_text(rsp):    return clean_response(rsp)
Powered by  Granite
```

Core tasks

The following 5 action classes define the core tasks performed by the agents in this AI-powered PRD generation workflow:

1. WritePRD creates the PRD.
2. SavePRD saves the PRD to disk.
3. ConductResearch generates supporting research.
4. PerformReview reviews the PRD.
5. RevisePRD revises the PRD based on feedback.

Define agent roles


Here are the agent roles that represent the multiagent PRD team. Below is the code that specifies which actions they perform.

```
class ProjectManager(Role):    name: str = "Pam"    profile: str = "Project Manager"    def
__init__(self, **kwargs):    super().__init__(**kwargs)    self.write_action = WritePRD()
self.save_action = SavePRD()    self.revise_action = RevisePRD()
self._watch([UserRequirement, ConductResearch, PerformReview])
self.set_actions([self.write_action, self.save_action, self.revise_action])    async def _act(self) ->
Message:    logger.info(f"{self.profile}: Starting PRD generation process.")    memories =
self.get_memories()    # If this is the first round, generate and save the draft PRD    if not
any(m.role == "Researcher" or m.role == "Reviewer" for m in memories):    msg =
self.get_memories(k=1)[0]    prd_content = await self.write_action.run(msg.content)
draft_save_result = await self.save_action.run(prd_content, filename="DraftPRD.md")    return
Message(    content=draft_save_result,    role=self.profile,
cause_by=type(self.write_action)    )    # If this is the second round, combine revised PRD and
research, then save    else:    research_msgs = [m for m in memories if m.role ==
"Researcher"]    review_msgs = [m for m in memories if m.role == "Reviewer"]
research_content = research_msgs[-1].content if research_msgs else "No research found."
review_content = review_msgs[-1].content if review_msgs else "No review found."    # Load the
draft PRD from file or memory    with open("DraftPRD.md", "r", encoding="utf-8") as f:
prd_content = f.read()    # Only revise if review feedback exists and is not empty
if review_msgs and review_content.strip() and review_content.strip() != "No PRD draft
found.":    revised_prd = await self.revise_action.run(prd_content, review_content)
else:    logger.info(f"{self.profile}: No review feedback found, skipping revision this round.")
revised_prd = prd_content.strip()    final_content = (    f"{revised_prd}\n\n"
f"---\n\n"    f"## Research\n{research_content}\n"    )    await
self.save_action.run(final_content, filename="PRD.md")    return Message(
content=final_content, # Only the markdown document    role=self.profile,
cause_by=type(self.write_action)    )
class Reviewer(Role):    name: str = "Rico"    profile: str =
"Reviewer"    def __init__(self, **kwargs):    super().__init__(**kwargs)    self.review_action =
PerformReview()    self.set_actions([self.review_action])    self._watch([WritePRD])    async
def _act(self) -> Message:    try:    with open("DraftPRD.md", "r", encoding="utf-8") as f:
prd_content = f.read()    except FileNotFoundError:    prd_content = "No PRD draft
found."    logger.info(f"{self.profile}: Reviewing PRD...")    review_content = await
self.review_action.run(prd_content)    return Message(content=review_content, role=self.profile,
cause_by=type(self.review_action))
class Researcher(Role):    name: str = "Rita"    profile: str =
"Researcher"    def __init__(self, **kwargs):    super().__init__(**kwargs)
```

```

self.research_action = ConductResearch()    self.set_actions([self.research_action])
self._watch([UserRequirement, WritePRD])    async def _act(self) -> Message:    try:    with
open("DraftPRD.md", "r", encoding="utf-8") as f:    prd_content = f.read()    except
FileNotFoundError:    prd_content = "No PRD draft found."    logger.info(f"{self.profile}:
Researching for PRD...")    research_content = await self.research_action.run(    "Provide
supporting research for the following PRD.", context=prd_content    )    return
Message(content=research_content, role=self.profile, cause_by=type(self.research_action))

```

Powered by  Granite

Core workflow and role definitions

The following agents collaborate to automate each step of the PRD creation process:

- ProjectManager (Pam) creates, saves and revises the PRD.
- Reviewer (Rico) reviews the PRD and provides feedback.
- Researcher (Rita) generates supporting research for the PRD.


Step 8. Assemble and run the Team

Use the Team class to hire agents and run the workflow. This app uses Typer to run the process interactively from the command line.

```

import typer import asyncio from metagpt.team import Team app = typer.Typer() @app.command() def
main(    idea: str = typer.Argument(..., help="A PRD for a banking application for wealth
management"),    investment: float = typer.Option(3.0, "--investment", "-i", help="Dollar amount to
invest in the project."),    n_round: int = typer.Option(2, "--n-round", "-n", help="Number of rounds to
run the simulation."), ):    async def runner():    team = Team(use_mgx=False)    team.hire([
    ProjectManager(),    Researcher(),    Reviewer(),    ])    team.idea = idea
team.invest(investment=investment)    team.run_project(idea)    await
team.run(n_round=n_round)    asyncio.run(runner()) if __name__ == "__main__":    app()

```

Powered by  Granite

How the CLI works

- @app.command() : Tells Typer that the following function (main) is a command that can be run from the terminal.
- main(...) : The main function that runs the program. It takes three arguments:
 - idea : The project idea (for example, “A PRD for a banking application for wealth management.”)
 - investment : An optional investment amount (default:3.0). This action simulates the team’s budget and can affect agent decision-making and planning.
 - n_round : Optional argument specifying how many rounds the simulation should run (default:2).

Team assembly and execution

- runner() : An asynchronous function that runs the workflow:
 - team = Team(use_mgx=False) : Creates a new Team object representing a group of AI agents. The use_mgx=False option disables the advanced MGX communication mode by using standard team behavior.

- `team.hire([...])` : Hires (adds) the agents to the team.
- `team.idea = idea` : Sets the team's project idea from the CLI input.
- `team.invest(investment=investment)` : Allocates the team's "funding," influencing how agents simulate planning and resource allocation.
- `team.run_project(idea)` : Starts the project with the given idea.
- `await team.run(n_round=n_round)` : Runs the workflow for the specified number of rounds, allowing agents to iteratively improve the PRD.

Note on output variability

Important note:

The documents and outputs generated by this tutorial use large language models (LLMs), which are probabilistic and can occasionally produce incomplete, inaccurate or inconsistent results.

Always review and validate all generated content yourself.

LLMs are helpful tools, but cannot fully replace the expertise and judgment of a real product development team.

Example command

To run the program with default values for `n_round` and `investment` :

```
python metagpt_prd_generator.py "Write a PRD for a banking app for wealth managers."
```

Powered by  Granite

This command will launch the team of agents, automate the PRD creation process and iterate for the specified number of rounds.

Summary of team creation process

- **Actions** define what each agent can do.
- **Roles** represent agents and connect them to actions.
- The **Team** class brings agents together to automate the PRD workflow.

This modular approach allows room for fine-tuning the process of automating complex product development tasks.

Example output draft vs. final PRD

When you run the application, the agents collaborate to produce and refine a PRD.

- **Draft PRD** (`DraftPRD.md`) : The initial PRD created by the project manager agent.
- **Review feedback**: Suggestions and critiques from the reviewer agent.
- **Research report**: Supporting technical and market research from the researcher agent.
- **Final PRD** (`PRD.md`) : The revised PRD, incorporating review and research.

Draft PRD (excerpt)

```
# Product Requirements Document: Wealth Manager Banking App ## 1. Introduction & Overview *
**Product Name:** [Proposed Name - e.g., "WealthBank Pro", "Portfolio Navigator"] * **Version:**
v0.1 (Initial Draft) * **Author:** [Your Team/Name] * **Date:** October 26, 2 * **Status:**
Draft ## 2. Purpose & Goals The purpose of this app is to solve key pain points faced by wealth
```

managers and their clients with a seamless, integrated digital platform for managing assets, monitoring portfolio performance, accessing banking services, and facilitating communication within the financial advisory context. ... (see full draft (`example_DraftPRD.md`) for more sections)

Powered by  Granite

Final PRD (excerpt)

```
# Product Requirements Document: Wealth Manager Banking App ## 1. Introduction & Overview *
**Product Name:** WealthBank Pro (or Portfolio Navigator - to be confirmed) * **Version:** v0.2
* **Author:** [Your Team/Name] * **Date:** October 26, 2023 * **Status:** Draft *
**Document Revision Notes:** Addressed reviewer suggestions by clarifying terms, adding
measurable goals, expanding integrations, including user stories, and added missing sections (User
Roles, Data Flow). ## 2. Purpose & Goals This app provides a secure digital platform for financial
advisors to manage client portfolios and offers clients an intuitive interface to monitor their
investments alongside core banking services. #### Measurable Key Goals: 1. **Enhance Advisor
Efficiency:** Reduce investment monitoring time by [Specify %]%, decrease report generation time by
[Specify %]%. 2. **Improve Client Experience:** Achieve a Net Promoter Score (NPS) of [Target
NPS score], increase client engagement via app to [Target percentage]%. 3. **Secure Collaboration:**
Reduce email inquiries between advisor and clients by [Target reduction %]%, ensure all messages are
traceable within the platform. ## 3. User Roles * **Wealth Manager/Financial Advisor:** Full access
to assigned portfolios, reporting, and communication. * **High-Net-Worth Client:** View-only
access to their own portfolio and account information. * **Administrative Staff (Optional):**
Read-only access for reporting/client onboarding. ... (see full final PRD (`example_PRD.md`) for more
sections)
```

Powered by  Granite

Improvements made by the agents

The final PRD.md file includes a section titled **document revision notes** that summarizes the key changes made during the review and revision process. This section helps stakeholders quickly understand what was updated in the document.

Here are the main enhancements found in the final PRD for the wealth manager application:

- **Measurable goals added:** Added success metrics like clear KPIs including NPS scores and time reductions to inform MVP and optimization.
- **User roles:** Defined roles and permissions for advisors, clients and staff.
- **Integrations:** Specified APIs and security protocols for data flow.
- **Document revision notes:** Summarized key changes for easy tracking.
- **User stories:** Expanded and clarified scenarios for actionable requirements.
- **Research:** Incorporated go-to-market strategy like pricing models and technical data to support decisions.

Example output files

- DraftPRD.md : Initial requirements document.
- PRD.md : Final, revised requirements document.
- (Optional) Research.md : Market and technical research supporting the PRD.

Conclusion

By following this tutorial, you've learned how to automate the creation and refinement of a product requirements document by using MetaGPT and Ollama. You set up a multi-agent team, defined custom actions and roles and ran an iterative workflow that produces high-quality, actionable PRDs. This modular approach can be adapted for other collaborative AI tasks, making it a powerful tool to streamline AI product management.