Agents Project Progress Document

# Step 1 – Create the New Repo Folder

We created a new folder dedicated to the agents system to keep it isolated from existing repositories (upa-pool-league-mobile and upa-pool-league-backend). This serves as the root for all agent-related code, configs, prompts, and generated artifacts.  
  
Commands:  
mkdir agents-project  
cd agents-project  
  
Result: An empty 'agents-project' folder ready to host the template.

# Step 2 – Initialize Git and Add Base Files

We initialized Git to enable version control and created base files for documentation and ignore rules.  
  
Commands (copy-paste safe):  
cd ~/Documents/agents-project  
git init  
echo "# Agents Project" > README.md  
echo "\_\_pycache\_\_/\nvenv/\n.env\n.DS\_Store\n.idea/\n.vscode/\n\_\_pycache\_\_/\n\*.pyc\nartifacts/" > .gitignore  
  
Result: Clean Git repo with README.md and .gitignore in place.

# Step 3 – Create Base Folder Structure

We created a reusable directory layout that organizes configuration, prompts, schemas, ingestion, agent code, and outputs.

Commands (copy-paste safe):

cd ~/Documents/agents-project  
mkdir -p configs prompts schemas data storage agents pipelines integrations ingestion tests artifacts docs

Folder purposes:  
• configs – per-project YAML configs (e.g., upa.yaml)  
• prompts – role prompts (PRL/SRA/USG/QAT)  
• schemas – JSON/YAML schemas for outputs  
• data – local inputs (e.g., rule PDFs)  
• storage – vector DB/cache (gitignored later)  
• agents – Python modules for each agent  
• pipelines – orchestration graphs  
• integrations – adapters (GitHub/Slack)  
• ingestion – PDF→MD, embeddings, chunkers  
• tests – schema/contracts tests  
• artifacts – generated outputs per run  
• docs – living docs (changelog, decisions)

# Step 4 – Set up Python Virtual Environment & Requirements

We created a clean Python virtual environment and installed dependencies required for the agents system. This keeps everything isolated from other projects and ensures reproducibility.  
  
Commands (copy-paste safe):  
cd ~/Documents/agents-project  
python3 -m venv venv  
source venv/bin/activate  
echo "langgraph==0.2.33  
langchain==0.2.14  
boto3==1.34.162  
chromadb==0.5.5  
pypdf==4.2.0  
tqdm==4.66.4" > requirements.txt  
pip install -r requirements.txt  
  
Result: A virtual environment (venv) is created, and dependencies are installed:  
• langgraph – agent orchestration  
• langchain – retrieval and LLM helpers  
• boto3 – AWS Bedrock SDK  
• chromadb – local vector DB for embeddings  
• pypdf – PDF text extraction  
• tqdm – progress bars for ingestion tasks  
  
This sets a solid foundation for building and running agents.

## Step 4 – Verification & Troubleshooting (Run on your machine)

Run these commands to verify the virtual environment and dependencies are correctly installed. Compare your outputs to the expected results below.

Commands (copy-paste safe):  
cd ~/Documents/agents-project  
source venv/bin/activate  
python --version  
pip --version  
python -c "import langgraph, langchain, boto3, chromadb, pypdf, tqdm; print('OK')"

Expected:  
• python --version → prints a valid Python 3.x version (e.g., 3.11.x)  
• pip --version → shows the pip inside your venv path  
• import test → prints 'OK' with no errors

## Industrial-Grade Practices: Quality Gates Before Coding Agents

To align with big-tech engineering standards, we will layer quality gates incrementally:  
• Branching: 'main' (protected), 'dev' for integration, feature branches 'feat/<agent-name>'.  
• Conventional Commits: feat:, fix:, docs:, chore:, test:, refactor: for clean history and automated changelogs.  
• Tooling (next steps): ruff (lint), black (format), mypy (types), pytest (tests), pre-commit hooks, GitHub Actions CI.  
• Documentation: keep README.md and docs/ updated, and maintain this Word progress log.  
• Definition of Done: code + tests pass locally and in CI, docs updated, artifacts generated, and issues/PRs linked.

# Step 5 – Create Initial Project Config (configs/upa.yaml)

We created the first per-project configuration file. This file lets the agents run without touching other repos, and it’s reusable for future projects by changing values. The config defines project name, rule inputs, output paths, and optional integrations (GitHub/Slack).

Commands (copy-paste safe):

cd ~/Documents/agents-project  
cat > configs/upa.yaml << 'YAML'  
project\_name: "UPA Pool League"  
modes: ["8-ball", "9-ball", "10-ball"]  
inputs:  
 rules\_pdfs: [] # add local PDF paths later, e.g., ./data/upa\_8ball\_rules.pdf  
outputs\_dir: "./artifacts"  
github:  
 enabled: false # safe default; enable later when ready  
 owner: "your-github-username"  
 repo: "upa-pool-league-mobile"  
 labels: ["agent", "story", "upa"]  
slack:  
 enabled: false  
 webhook\_url: "${SLACK\_WEBHOOK:-}"  
models:  
 reasoning: "bedrock:anthropic.claude-3-5-sonnet" # placeholder; we will wire this later  
 embedding: "bedrock:amazon.titan-embed-text-v2"  
rag:  
 chunk\_size: 1200  
 chunk\_overlap: 150  
 collection: "rules"  
YAML

Result: A reusable configuration (configs/upa.yaml) now exists and will be passed to the pipeline entry point to control ingestion, model selection, and optional integrations. We keep GitHub/Slack disabled by default.

# Step 6 – Create Initial Prompts (Developer-Style, File-First)

Instead of shell commands, we’re creating files like a normal developer using an editor. Add the following four prompt files under the 'prompts/' folder. These define clear roles and output expectations.

## prompts/prl\_clarify.txt

You are the Product Requirements Liaison (PRL) for a pool league scoring app. Given a project plan and notes, produce a Clarification Pack with:  
1) Top ambiguities (numbered),  
2) Proposed assumptions (short),  
3) A 100–150 word summary.  
Cite rule sections if referenced. Keep tone concise.

## prompts/sra\_rules.txt

You convert rules text into machine-readable JSON with stable section IDs. For each rule, include: id, name, constraints, allowed\_values (if applicable), and notes (verbatim excerpt). Ensure IDs are stable (e.g., UPA-8B-3.3.4). Output valid JSON only.

## prompts/usg\_storysmith.txt

You transform CONFIRMED requirements into INVEST user stories. Output a JSON array. Each story includes: story\_id, as\_a, i\_want, so\_that, acceptance\_criteria[], rules[]. Acceptance criteria must be crisp and testable. Reference rule IDs from the SRA JSON.

## prompts/qat.txt

You convert acceptance criteria into actionable test outlines for a mobile app. Return a simple list of steps like 'Open Scoring screen -> set break type -> save -> verify'. Focus on clarity, determinism, and minimal steps to validate each AC.

Result: We now have role-specific prompt files that are reusable across projects and enforce consistent outputs.

# Step 7 – Add Pipeline Skeleton & Smoke Test (No Model Calls Yet)

We created a minimal orchestration pipeline that wires four agent stubs together (PRL → SRA → USG → QAT). This proves the end-to-end flow without relying on any external model calls or vector stores.

Files to add (edit in your IDE):

## pipelines/plan\_graph.py

Defines a tiny pipeline class and four stub stages. Each stage reads from and writes to a shared dict-like state. This mirrors our future LangGraph wiring but keeps it dependency-light for now.

## agents/prl.py

Contains a PRL stub function that simulates clarifications generation from a project plan input.

## agents/sra.py

Contains an SRA stub that returns a minimal machine-readable rules JSON structure.

## agents/usg.py

Contains a USG stub that converts confirmed requirements + rules into two example stories.

## agents/qat.py

Contains a QAT stub that turns acceptance criteria into a short test plan list.

## run.py

Entry point to run the pipeline in-process and print artifacts to the console. This is our first sanity check.

## tests/test\_pipeline\_smoke.py

A basic smoke test that imports the pipeline and verifies that all expected keys are produced in the final state.

Next step after adding these files: run the smoke test (`pytest -q`) and then execute `python run.py` to see the printed artifacts.

# Step 7a – Fix Test Imports (Make packages + pytest config)

Pytest couldn't import our modules because Python didn't see 'agents' and 'pipelines' as packages. We fix this by adding \_\_init\_\_.py files and configuring pytest to include the project root on PYTHONPATH.

Create these files (developer-style in your editor):

1) agents/\_\_init\_\_.py (empty file)

2) pipelines/\_\_init\_\_.py (empty file)

3) pytest.ini (at repo root):

[pytest]  
pythonpath = .

Now run tests again:  
source venv/bin/activate  
pytest -q  
  
Expected: tests collect and run without ImportError.

# Step 8 – Add Quality Gates (Ruff, Black, Mypy, Pytest, Pre-commit, CI)

We’re hardening the project like an industry setup: static analysis (ruff), formatting (black), type checks (mypy), tests (pytest), pre-commit hooks, and a basic GitHub Actions CI to enforce gates on each PR.

## A) Install dev tools (one at a time in your venv)

1) pip install ruff

2) pip install black

3) pip install mypy

4) pip install pytest (already used; keep pinned)

5) pip install pre-commit

## B) Create pyproject.toml at repo root

[tool.black]  
line-length = 100  
target-version = ["py311", "py313"]  
  
[tool.ruff]  
line-length = 100  
select = ["E", "F", "I", "UP", "B"]  
ignore = []  
target-version = "py311"  
exclude = ["venv", "storage", "artifacts"]  
  
[tool.mypy]  
python\_version = "3.11"  
strict = false  
ignore\_missing\_imports = true  
exclude = "venv|artifacts|storage"

## C) Create .pre-commit-config.yaml

repos:  
- repo: https://github.com/astral-sh/ruff-pre-commit  
 rev: v0.6.8  
 hooks:  
 - id: ruff  
 args: ["--fix"]  
- repo: https://github.com/psf/black  
 rev: 24.8.0  
 hooks:  
 - id: black  
- repo: https://github.com/pre-commit/mirrors-mypy  
 rev: v1.10.0  
 hooks:  
 - id: mypy  
- repo: https://github.com/pre-commit/pre-commit-hooks  
 rev: v4.6.0  
 hooks:  
 - id: end-of-file-fixer  
 - id: trailing-whitespace

Then enable hooks:  
pre-commit install  
pre-commit run --all-files

## D) Optional: .editorconfig (consistent editors)

root = true  
  
[\*]  
charset = utf-8  
end\_of\_line = lf  
insert\_final\_newline = true  
indent\_style = space  
indent\_size = 2

## E) Add CI: .github/workflows/ci.yml

name: CI  
on:  
 pull\_request:  
 push:  
 branches: [ main, dev ]  
jobs:  
 build:  
 runs-on: ubuntu-latest  
 steps:  
 - uses: actions/checkout@v4  
 - uses: actions/setup-python@v5  
 with:  
 python-version: '3.13'  
 - name: Install deps  
 run: |  
 python -m pip install --upgrade pip  
 pip install -r requirements.txt  
 pip install ruff black mypy pytest  
 - name: Lint (ruff)  
 run: ruff check .  
 - name: Format check (black)  
 run: black --check .  
 - name: Type check (mypy)  
 run: mypy .  
 - name: Tests (pytest)  
 run: pytest -q

## F) How we use these gates

• Local dev: pre-commit blocks bad diffs; you run tests before commit.  
• CI: enforces ruff/black/mypy/pytest on PRs to main/dev.  
• Definition of Done: code passes gates locally + CI, docs updated, artifacts generated.

Next: Step 9 will add the first real ingestion function (PDF→Markdown) and a tiny RAG store (Chroma).

# Boilerplate Suitability & Portability Assessment

Short answer: Yes. This agents setup is a reusable boilerplate. With minor renaming and a new project config, you can clone it for future projects without touching the architecture. Below is what to change for a new project and why the setup works.

## What to change for a new project (minimal renames)

1) configs/<project>.yaml – Create a new YAML (e.g., configs/myapp.yaml) with project\_name, inputs, outputs\_dir, and integrations.  
2) README.md – Update the title and one-sentence description.  
3) GitHub integration (optional) – In configs/<project>.yaml, set github.owner, github.repo, labels.  
4) Prompts (optional) – If the domain changes, tweak terms in prompts/\*.txt (e.g., PRL mentions 'PO' vs. 'stakeholder').  
5) Rules ingestion (domain-specific) – Point inputs.rules\_pdfs to your domain documents.  
6) CI badge (optional) – Update README badge once the repo is on GitHub.

## Why this setup works as a boilerplate

• Separation of concerns – configs, prompts, agents, pipelines, ingestion, and integrations are isolated.  
• Schema-first outputs – stories/rules/testplans are consistent and machine-consumable.  
• Artifact-first – each run writes to /artifacts with timestamps, enabling easy review and diffs.  
• No hard coupling – integrations are adapters; if disabled, the system runs fully offline.  
• Testable from day 1 – smoke tests verify the pipeline; we add more tests as features land.  
• Quality gates – ruff/black/mypy/pytest + pre-commit + CI enforce standards on every PR.

## Python & tooling compatibility notes

• You’re on Python 3.13.2. Update tool targets to match your runtime:  
 - In pyproject.toml, set [tool.black].target-version to ["py313"] and [tool.ruff].target-version = "py313".  
 - In [tool.mypy], set python\_version = "3.13".  
• Pinned dev tools in requirements.txt ensure reproducible installs; we can move dev-only pins to a separate dev-requirements.txt later if desired.

## Portability checklist (for any new project)

□ Copy the repo (or use it as a GitHub Template).  
□ Add configs/<new>.yaml with project\_name and inputs.  
□ Drop domain docs into /data and reference them in the config.  
□ Run python run.py with --config pointing to the new YAML (we’ll add the flag in a later step).  
□ If using GitHub issues, add a PAT and flip github.enabled: true in the config.  
□ Ensure CI badge and branch protections are set up in the new repo.

## Any caveats?

• Domain rules differ: SRA prompts or parsing may need small tweaks for non-UPA rule formats.  
• Tooling updates: Dev tool versions evolve; we pinned versions to stabilize, but revisit quarterly.  
• Secrets: Keep tokens in .env or GitHub Actions secrets, never in repo.  
• Data volume: If docs get large, we’ll swap the local vector store to Postgres/pgvector.

Conclusion: The current structure is a solid, industrial-grade boilerplate. Rename the config, adjust prompts minimally, and it’s ready for any future project without rewiring.