ChatGPT

Agent-Driven Autonomous Development Framework

Modern AI agent architectures enable entire software development teams to be run by intelligent agents [1]. In our framework, each agent is a stateful LLM-powered process with a specific role (e.g. Planner, Coder, Tester). We adopt patterns from production-ready agent stacks: for example, LangChain treats LLMs as autonomous "agents" that decide actions based on context [2], and systems like CrewAI show how to coordinate a "dream team" of role-playing AI agents [3]. Agents communicate via messages or APIs (e.g. a Redis queue or HTTP endpoints), collaborating on tasks. Over time the agent "company" self-improves by learning from outcomes: successes and failures are fed back into its memory to refine behavior.

Knowledge Base & Memory System

A persistent knowledge base is critical so agents don't suffer "digital amnesia." We implement **long-term memory** with a vectorized database and document store [4]. All project documentation, code history, design notes, meeting transcripts, and past agent outputs are indexed (for example by OpenAI or MiniLM embeddings) into a local vector DB (such as Redis with vector support or Chroma) [4] [5]. At runtime, agents use Retrieval-Augmented Generation (RAG): they query the vector store to fetch relevant facts or past decisions when solving a task [4]. One tutorial shows a dual-memory agent using Redis: a short-term cache plus a long-term vector store, enabling the agent to remember user preferences and conversation history [5]. In our design, **short-term memory** is simply the current prompt/context, while **long-term memory** is this searchable knowledge base. By persistently saving every design rationale and code snippet, agents can recall and build on earlier work.

- *Short-term context:* Rolling conversation or task context (token window).
- *Long-term memory:* A database (files + vectors) for codebase and domain knowledge. We can use Chroma (an open-source AI vector DB that "works locally" [6]) or Redis.
- *Knowledge augmentation:* We also consider knowledge graphs for complex relationships, but start with vectors. IBM notes that long-term AI memory is often built with databases or vector embeddings [4].

Technology Stack (Local-First)

All components must run on standard VPS hardware. We propose an open-source stack:

- **Agent Framework:** LangChain (Python) with LangGraph (for stateful workflows). LangChain provides chains, memory support, and tools, and treats LLMs as true agents [2].
- **LLM Model:** An LLM (cloud or local). For local use, we run quantized open models (e.g. Llama 2 7B) using C++/GGML libraries or Llama.cpp. The repository *Llama-2-Open-Source-LLM-CPU-Inference* shows Llama-2-7B-Chat running on CPU with 4-bit quantization [7]. We use Llama2 (7B or 13B) for code generation and planning, and smaller embedding models (e.g. MiniLM) for vectorizing text [7].

- **Vector Database:** A local vector store for RAG. Options include **Chroma** (pip-installable, Apache-2.0, designed for local AI apps [6]) or Redis with vector extension [5]. Chroma's docs explicitly say "Develop locally" for AI search [6]. We store embeddings for project docs, code, PR history, etc.
- **Knowledge Base:** A file database (e.g. markdown docs or SQLite) plus the vector index. For example, we keep design docs and API references in files and index them. Retrieval from the vector DB can fetch relevant docs or code for the agent's context [4].
- **Tooling & Integrations:** Agents may call external tools (e.g. `git`, linters, compilers) via LangChain tool interfaces. We integrate common dev tools for building/testing.
- **Web/API Interface:** We expose agents as services. For example, use FastAPI to create REST or streaming endpoints [8]. This lets humans submit tasks or queries. A simple UI (Streamlit chat or React dashboard) can display agent suggestions and logs [9]. NirDiamant's tutorials include a Streamlit chatbot example for agent demos [9].
- **Deployment:** Containerize with Docker for portability [10]. Each agent service (and DB) runs in its own container. We host everything on a classic Linux VPS. To avoid cloud LLM APIs, we can deploy an on-prem model serving solution (e.g. Ollama) to run LLMs locally [11]. GPU scaling isn't assumed; instead we rely on optimized CPU inference (4-bit models) and efficient batching.

Multi-Agent Coordination

Agents are organized like a development team. We designate a **Coordinator** agent that decomposes high-level goals into sub-tasks and assigns them. For example, a Planner agent reads a feature spec and creates a work breakdown (in PRP form). Then Coder agents implement code, Tester agents run tests, and DevOps agents deploy. Agents communicate via a shared message bus (Redis pub/sub) or via HTTP callbacks. We can leverage multi-agent frameworks: for instance, the open Coral protocol supports autonomous agent collaboration [12]. Our design also draws on CrewAI's insight: treat the agent team as a group of domain experts, each with specialized skills [3]. By structuring agents and tasks clearly, we enable parallelism (multiple issues or components worked on concurrently). Over time, agents coordinate better – successes and metrics are logged, so agents adjust (e.g. prefer more reliable tools or retry failed steps).

- **Agent Roles:** Example roles include *Planner* (task breakdown, writing PRPs), *Coder* (writes code or scripts), *Reviewer* (runs tests or code analysis), *Integrator* (merges changes, manages version control).
- **Communication:** Use Redis (or HTTP) for passing task messages. Could adopt a multi-agent protocol (like A2A) or a lightweight pub/sub.
- **Monitoring:** All actions are logged. We plan to integrate observability tools (per Diamant's tutorial on Qualifire) to trace agent decisions and performance.

Initial Project Plan (PRP Methodology)

We frame the development plan as a **Product Requirement Prompt (PRP)** [13]. A PRP includes goal, justification, context, implementation steps, and validation. For example:

- **Goal:** Build an autonomous development platform powered by AI agents.
- **Why:** To automate software creation and speed up delivery. Agents can work continuously without human fatigue, enabling rapid project growth.
- **Context:** We base our architecture on existing agent guides. Diamant's "Agents Towards Production" repo provides tutorials on memory, tool integration, and orchestration [1]. We have local files (code,

docs) and a vector DB for context. Each new feature will be specified in a PRP that includes the business need and relevant code snippets (as per the PRP approach [13]).

- **Implementation Blueprint:**
- **Setup Infrastructure:** Provision a VPS and install Python, Docker, and databases. Install LangChain, FastAPI, and the vector DB (e.g. Chroma). Download or prepare LLM models (e.g. Llama2 in quantized format) for inference [7].
- **Initialize Knowledge Base:** Ingest initial project data (requirements docs, previous code, architecture notes) into the vector store. Verify that agents can retrieve facts from it.
- **Define Agents and PRPs:** Create prompt templates for agents. For each role, set up a PRP template (with fields: Goal, Why, Context, Tasks, Tests). For example, a PRP for "User authentication" would list the goal ("implement login"), relevant code files, and acceptance tests [13].
- **Agent Workflows:** Implement an interactive loop (or use PRP scripts) that feeds each PRP to the appropriate agent. For instance, run `/create-base-prp` (via Claude or ChatGPT) to craft detailed tasks, then `/execute-base-prp` to have the Coder agent write code. After coding, the Reviewer agent runs tests (e.g. via pytest) and reports results. Use a combination of CLI scripts and API calls to automate this flow.
- **Validation Loop:** For each completed PRP, automatically run the validation (unit/integration tests). Failed tests generate new PRPs to fix bugs. This feedback is stored in memory (e.g. "fixed bug X in module Y"), so the system learns from mistakes.

- **Iterate and Improve:** Use monitoring to track metrics (test pass rates, time per task). Periodically review and refine agent prompts and knowledge. Agents update the knowledge base with key learnings (e.g. code snippets that solved a problem).

- **Validation Criteria:** We will know success when the system can autonomously deliver a small web app from spec to deployment. For example, giving it a feature list and getting fully tested, running code back (with Git commits). Key metrics include correctness (passing tests), efficiency, and "time-to-delivery" improvements. Each PRP-driven iteration is assessed by its test results – agents are rewarded or constrained accordingly.

By using structured PRPs, each agent has a clear, self-contained specification to act on [13]. This method ensures that requirements, code context, and verification steps are all fed to the AI. As a result, the "company" grows: completed tasks augment the knowledge base, making future PRPs more informed (agents remember code conventions, past bugs, etc. via the vector memory [4]).

Sources: This design is informed by the *Agents Towards Production* tutorials and open AI engineering guides [1] [5], which cover memory systems and agent orchestration; by LangChain and agent framework literature [2] [3]; and by AI memory best practices [4]. These references underpin our choice of tools (LangChain, local LLMs, Chroma/Redis) and methodology (PRP-driven development).

---

[1] [5] [8] [9] [10] [11] [12] GitHub - NirDiamant/agents-towards-production: This repository delivers end-to-end, code-first tutorials covering every layer of production-grade GenAI agents, guiding you from spark to scale with proven patterns and reusable blueprints for real-world launches.
https://github.com/NirDiamant/agents-towards-production

[2] [3] LLMs, Agents, Multi-agents… and now, self-improving ones | by Diego Romero | Medium
https://dromerosm.medium.com/llms-agents-multi-agents-and-now-self-improving-ones-1227b33e0262

[4] What Is AI Agent Memory? | IBM
https://www.ibm.com/think/topics/ai-agent-memory

[6] Chroma
https://www.trychroma.com/

[7] GitHub - kennethleungty/Llama-2-Open-Source-LLM-CPU-Inference: Running Llama 2 and other Open-Source LLMs on CPU Inference Locally for Document Q&A
https://github.com/kennethleungty/Llama-2-Open-Source-LLM-CPU-Inference

[13] README.md
https://github.com/Wirasm/PRPs-agentic-eng/blob/0c7a69c7e38d52d02a94ef93a92c3c980157da68/README.md