ChatGPT

Designing a Rust-Based Swarm of Collaborative Agents

Goals and Requirements

- **Ephemeral Agent Swarm:** The system can **spin up a swarm of agents** dynamically to tackle a given problem, and terminate them once the problem is solved. This ensures resources are freed after use.
- **Parallel & Distributed Operation:** Agents should run **in parallel** on a single machine (leveraging multi-core CPUs) and be deployable across a cluster for scalability. Using containerization (Docker) and orchestration (Kubernetes) can enable distributing agents across machines for heavy workloads.
- **Secure and Safe Execution:** Communications and code execution must be secure. Agents should communicate over encrypted channels when distributed, and any code they generate/run should be sandboxed to prevent security breaches. Rust's memory safety helps prevent common exploits (e.g. code injection) and sandboxing issues [1].
- **Memory Management:** Provide both **shared memory** (common knowledge store for collaboration) and **isolated memory** per agent (for private computations). Include short-term memory for the current task context and optional long-term memory for persistence. Agents need a mechanism to exchange information (e.g. a shared blackboard or message-passing bus).
- **Resource Efficiency:** The framework should be lightweight and run in low-resource environments (CPU-friendly). Rust's efficiency (compiled code, no heavy runtime) ensures minimal overhead compared to scripting languages (avoiding issues like the GIL in Python) [2]. The entire system can be delivered as a single static binary for ease of deployment [3].
- **Flexible Problem Solving:** Although code generation (e.g. building an API) is a primary use-case, the system should handle *any coding or data-processing task*. Agents may be specialized (code writer, tester, data pipeline runner, documentation generator, etc.) but the framework should allow configuration of roles per user needs.
- **Extensible Tools Integration:** Users should be able to **add external tools or services** that agents can utilize (for example, an API to fetch data, a compiler, or a long-term memory store). The design should support plugging in these tools, including integration with **MCP (Model Context Protocol)** servers for extended memory or knowledge bases [4]. Using the MCP standard means the agents can securely access external context and services without extensive custom integration code [4] [5].

System Architecture Overview

The high-level architecture will consist of a **central orchestrator** and multiple **agent instances** (the swarm). This follows a *centralized* multi-agent architecture where a controller coordinates the agents [6], which simplifies implementation and ensures consistent operation. Key components of the system include:

- **Orchestrator (Control Unit):** A central component that manages the lifecycle of agents and task scheduling. It interprets the user's problem input and launches the necessary agents. The orchestrator assigns tasks to agents, monitors progress, and decides when to spawn or shut down agents. This aligns with the classic control unit in a blackboard system, which selects which knowledge source (agent) to execute based on the problem state [7].

- **Agents (Knowledge Sources):** A collection of autonomous agents, each with a specialized role or capability. For example, one agent might handle planning, others handle coding different modules, another does testing, etc. Each agent is an independent module (possibly an async task, thread, or separate process) that contributes its expertise to the problem [8]. They operate semi-autonomously but all work toward the common goal.
- **Shared Environment / Blackboard:** A **shared context or memory space** accessible by all agents. This can be implemented as a *blackboard*, i.e. a global data store (in-memory database, append-only log, or state object) where agents post their outputs and read others' contributions. The blackboard holds the problem state, partial solutions, and decisions made so far [9] [10]. All agent communication can occur through this shared medium – agents write to and read from the blackboard instead of direct peer-to-peer chats. This approach ensures every agent has access to the **full context and history** of the problem at all times [11], which is crucial for coherence. (For instance, a testing agent can see the code that coding agents produced by reading the blackboard.)
- **Communication Layer:** The mechanism by which agents and the orchestrator exchange information. In a local single-process deployment, this could be implemented with Rust channels or an internal pub-sub message bus. In a distributed setting, it could be a lightweight network API (e.g. gRPC or WebSocket servers for agents) that the orchestrator uses to dispatch tasks and collect results. Communication can follow a **publish/subscribe** or **direct messaging** model, but using the blackboard pattern effectively creates a **shared memory** model where posting to the blackboard is akin to broadcasting updates [12]. For targeted messages (e.g. orchestrator instructing a specific agent), direct RPC calls can be used. All network communication should use encryption (TLS) for security if agents are spread across machines.
- **Knowledge Base (Memory):** A system for both shared and private memory. The **shared memory** is primarily the blackboard (public space) containing all accumulated knowledge and intermediate results that every agent can access [13]. This shared memory equates to a global knowledge base for the task. In addition, each agent can maintain its **isolated memory** – this could be a small private context or scratchpad not posted to the blackboard unless needed. For example, an agent might keep a private copy of its ongoing thought process or use a private workspace for drafting code before sharing. Long-term memory (persisting beyond the current problem) can be implemented via an external database or an MCP-compatible memory server. By integrating an MCP server for memory, agents can store and retrieve long-term context (e.g. past project data, conversation history) using a standardized interface [14] [15]. *Short-term memory* (the current task context) lives in the blackboard and agent state, whereas *long-term memory* (knowledge from past tasks or domain info) can be fetched via tools like an MCP memory server or embedded local database.

All these components work together as follows: The **orchestrator (control unit)** monitors the content of the shared memory and overall task progress, deciding which agent(s) to activate at each stage [7]. Agents, upon activation, **perceive** the current state (by reading the blackboard or receiving a message with relevant context), **reason/plan** based on their role, then **act** by writing results or suggestions back to the shared memory (or sending a direct result message to the orchestrator) [7] [16]. This loop continues—possibly in several rounds or in parallel branches—until the problem is solved and the final solution is written to the blackboard for collection.

Agent Roles and Responsibilities

Each agent in the swarm will have a defined **role** with specific responsibilities, ensuring a divide-and-conquer strategy for problem solving. Roles can be fixed or configurable per the user's needs. For a coding task (like *"build an API for an app"*), one might use roles such as:

- **Planner Agent:** An agent that analyzes the high-level request and breaks it into sub-tasks or modules. For an API example, the planner might define what endpoints or components are needed and post this plan to the shared memory (e.g. a list of functions or classes to implement). This agent essentially coordinates the *strategy*.
- **Coding/Developer Agents:** One or multiple agents that take on the programming tasks. They might each pick up a specific sub-task from the plan (for example, one agent implements the user authentication module, another the database interface, etc.). Their goal is to produce code. Internally, a coding agent could leverage an LLM (Large Language Model) or code generation library to actually write the code based on specifications. Each coding agent writes its code output to the shared memory (or a shared code repository) so others can see it.
- **Testing Agent:** This agent is responsible for quality assurance. It monitors the code outputs and can generate tests or run existing tests on the code. If run-time verification is needed, the testing agent might compile and execute the generated code in a sandbox environment, then share the results (e.g. test passed/failed, error logs) on the blackboard. Any issues detected can be fed back into the loop – for instance, the testing agent can flag an error, and a coding agent might pick that up and fix the code.
- **Documentation Agent:** Once code is being produced, a documentation agent can create documentation or comments. It could read the code from the shared memory and produce README content or API docs for it. This agent ensures the final output includes human-friendly documentation.
- **Coordinator/Reviewer Agent:** (Optional) Another specialized role can be a reviewer or integrator that ensures all pieces produced by other agents fit together. It might review code style, integrate modules, or finalize the solution for delivery. In many cases the orchestrator itself can perform final integration, but a dedicated agent could handle complex integration logic.

These roles collaborate towards the common goal. They are **not siloed**; every agent's contributions become visible in the common environment so that, for example, the Documentation agent can see the code that the Developer agents wrote, and the Testing agent can run tests on that code. The design should allow **configurable roles** – e.g. if the user wants to add a "Data Cleaning Agent" or a "UX Design Agent" for different kinds of tasks, they can do so. New agent types can be added by implementing the agent trait (interface) and plugging them into the orchestrator's configuration.

Each agent will have a **defined interface** or trait in Rust. For instance, we might define a trait like:

```rust
trait Agent {
    fn name(&self) -> &'static str;
    fn role_description(&self) -> &'static str;
    fn execute(&mut self, shared_state: &SharedMemory) -> AgentOutput;
}
```

Each agent implementation would encapsulate the logic needed for its role. The `execute` method would contain the agent's strategy: e.g., a CodingAgent might take a spec from shared memory and call an LLM API to generate code, whereas a TestingAgent might retrieve the latest code and run a test suite. Agents could run **concurrently** in separate threads or async tasks if their tasks are independent. For example, multiple CodingAgents could implement different modules in parallel to speed up development. The orchestrator ensures that prerequisites for each agent are met (it might start the TestingAgent only after some code is available, for instance).

Communication and Coordination

Shared Blackboard Mechanism: The primary mode of coordination will be via the shared memory (blackboard). The orchestrator and all agents can read and write to this common space. This has several benefits:

- **Complete Visibility:** All agents see the big picture – the entire history of messages, plans, code snippets, decisions, etc., is available [11]. This reduces duplication of effort and miscommunication.
- **Loose Coupling:** Agents don't call each other directly (which would create complex interdependencies); instead they communicate through the blackboard, which decouples their interactions [16]. An agent doesn't need to know *which* agent will handle a sub-problem, it just posts the sub-problem to the board.
- **Conditional Action:** The orchestrator (or a control logic) can choose which agent to trigger based on the content on the board [7]. For instance, once code appears on the blackboard, the orchestrator can signal the TestingAgent to run tests. This is analogous to a **control unit** scanning the blackboard and scheduling the next knowledge source to run, as described in blackboard system architectures [7].

In practice, the blackboard might be represented by a data structure (like a thread-safe `Arc<Mutex<SharedState>>` in Rust for in-process, or a small database service for distributed mode). It could contain sections: e.g., a section for the task plan, a section for code outputs (possibly a git repository or just text blobs), a section for test results, etc. Agents will write entries to it (with timestamps or versioning as needed), and can also **subscribe** to changes. For example, a CodingAgent could subscribe to new task assignments added by the Planner.

Message Passing: In addition to the blackboard, direct message passing can be used when appropriate. The orchestrator might send direct commands to an agent (e.g., "Agent3, start coding module X now"), which could be done via an async channel or network message. If agents need to notify a specific agent privately, the system could support direct messages, but this should be minimized in favor of shared knowledge (to maintain transparency). In a distributed context, an internal **Pub/Sub** system can be set up where agents subscribe to certain topics (like "tests" or "docs") and get notified when relevant info is available [12]. Technologies like **NATS** or **Redis PubSub** could be utilized for a lightweight distributed messaging backbone, or simply use Kubernetes Services with HTTP/gRPC calls to each agent.

Secure Communication: When running on a single machine, agents communicate through in-memory channels (which are inherently secure from outside access). In a multi-node cluster, all RPC or message traffic should be encrypted. We can use mTLS (mutual TLS) between services or rely on Kubernetes' cluster network policies for isolation. Every agent service might have an authentication token to ensure only authorized orchestrator or peer agents can communicate with it. This prevents malicious interference or eavesdropping on agent messages.

Coordination Strategy: We will implement a coordination loop roughly as follows:

1. **Task Initialization:** The orchestrator populates the blackboard with the initial problem description and possibly an initial plan (if simple), then triggers the Planner agent (if one is used).
2. **Planning Phase:** The Planner writes a breakdown of tasks to the blackboard (e.g., list of modules or steps). It might tag each sub-task with recommended agent types (e.g., “Coding” or “DataFetch”).
3. **Task Assignment:** The orchestrator reads the plan and launches the required number of agent instances for each role (if not already running). Alternatively, some agents could be pre-started and waiting idle. The orchestrator either assigns sub-tasks directly to specific agents (e.g., sends a message “AgentX, do Task1”) or just relies on agents to pick tasks from the blackboard (agents can look for unclaimed tasks on the board and claim them).
4. **Parallel Execution:** Multiple CodingAgents (or others) work in parallel on their respective tasks. Each agent writes interim results to the blackboard. For example, a CodingAgent after completing a function posts the code on the board (or commits to a shared repository and notes it on the board).
5. **Synchronization:** As soon as a piece of work is done and posted, other agents react. The TestingAgent, upon seeing new code on the board, can retrieve it and run tests. If tests pass, it may post a confirmation; if they fail, it posts error logs or feedback.
6. **Iterative Refinement:** If any agent (or the orchestrator) notices an issue (like a test failure or a logical inconsistency in the plan), it can trigger a refinement. For instance, a Critic agent (if included) could analyze outputs and suggest improvements or catch errors. The responsible agent (e.g., a CodingAgent for a failed test) will then revise its output. This loop continues until consensus or solution is reached – meaning no further changes needed and all tests pass.
7. **Completion:** The orchestrator monitors if the overall goal is achieved (e.g., all required modules coded, all tests passing, documentation prepared). It then aggregates the results (final code, documentation, etc.) from the blackboard. Agents are instructed to shut down (or they self-terminate after completing their part). The orchestrator presents the final solution to the user and then can flush the shared memory (since the swarm’s life ends with the problem).

This mechanism ensures **efficient collaboration**: Agents are not idle waiting for one another more than necessary; many can work in parallel, yet the shared memory keeps their efforts aligned. The design also supports a degree of fault-tolerance: if one agent crashes or fails, the orchestrator can detect this (lack of heartbeat) and spin up a replacement agent, since all state is on the blackboard (the new agent can read the latest state and resume). This stateless-agent model (state in environment, not in the agent) draws from known multi-agent design principles where the environment holds the authoritative state [8].

Memory Management Strategy

As noted, memory is split into **short-term shared context** and **long-term knowledge**:

- **Shared Short-Term Memory:** The blackboard is the primary short-term memory. It contains all the data about the current problem – effectively the “working memory” of the agent swarm. Agents do not have to store large context locally; they pull what they need from the blackboard when executing. For example, before generating code, a CodingAgent will read the latest requirements or designs from the board. This shared memory contains **all historical context** of the problem-solving session (messages, partial solutions, outcomes) [13], so agents don't need separate conversation histories. (If using LLMs for agent reasoning, the prompt given to the LLM can be dynamically constructed from the blackboard contents relevant to that agent's task.)

- **Private Short-Term Memory:** In some cases it's useful for an agent to have a private scratchpad. For instance, an agent might be running an LLM chain internally and not all intermediate LLM thoughts need to be posted publicly until they're confirmed. An agent's isolated memory could simply be local variables or an internal struct that isn't shared. The design should *allow* this (for agent-specific computations), but ultimately any result that matters to the overall problem should be written to the shared memory so others can benefit. The blackboard architecture even allows **private sections** if needed (areas only certain agents can access) [16] [13], though initially we might keep everything public for simplicity.
- **Long-Term Memory:** For extended knowledge that persists across problems or provides context from outside the current problem, we integrate a long-term memory store. This could be a vector database, a file repository, or an **MCP server** providing semantic search over past data. **Model Context Protocol (MCP)** is ideal for plugging in such memory: the agent swarm can include an MCP *client* component that connects to an external MCP *server* (such as Pieces or Mem0) which holds persistent knowledge. For example, an agent can query an MCP server for "past design decisions" or "domain-specific info" relevant to the task, and incorporate that into its reasoning. MCP is designed to **let AI agents fetch external context securely and uniformly** [4]. By supporting MCP, users could hook up services like knowledge bases, documentation wikis, codebase indexes, etc., without custom integration (MCP provides a standard API for these) [4] [5].
- **Memory Implementation:** In Rust, the shared memory can be a globally accessible structure protected by synchronization (e.g., using `Arc<RwLock<...>>` or actor messages). The data might include structured entries (we could define types like `enum BlackboardEntry { Plan(...), CodeSnippet(...), TestResult(...) }`). Agents can filter relevant entries by type or tags. For long-term memory, the system might provide a helper module that interfaces with an external database. If an MCP server is used, the Rust orchestrator could make HTTP calls or use an SDK to query it and then feed the results into the blackboard for agents to use. This separation ensures that heavy or persistent data is not kept in RAM unnecessarily – it can be fetched on demand.
- **Garbage Collection and Limits:** Since tasks may produce a lot of data on the blackboard, we should have policies for memory management. The orchestrator can clean up intermediate data that is superseded by newer results (or keep a full history if needed for traceability). If running on limited memory, the blackboard could offload older entries to disk. Long-term memory queries should also be used sparingly (e.g., only when an agent specifically needs more context) to remain efficient with tokens if using LLMs.

In summary, the memory architecture combines a **global shared store** (like a **blackboard**, acting as the short-term memory and communication medium) with optional **persistent memory integration** for long-term knowledge. This approach is common in advanced multi-agent systems, where a knowledge base can be both shared and local [8] – here the blackboard is the shared knowledge base, and each agent's internal state is its local memory. This design ensures agents have a common frame of reference while still allowing private computation when needed.

Security and Isolation Considerations

Security is a first-class concern given that agents may execute code or handle sensitive data:

- **Memory Safety and Rust Guarantees:** By implementing the system in Rust, we inherently avoid many memory safety issues (buffer overflows, use-after-free, etc.). Rust's safety and error-handling features help prevent crashes and undefined behavior [1]. This is important if agents are running untrusted or AI-generated code – any rogue behavior is confined by Rust's safety (and by the sandboxing described below). Rust is also **fast and efficient**, meaning we can enable security features without huge performance loss. The choice of Rust gives us a "secure-by-design" foundation for the agent framework [17].
- **Sandboxing Code Execution:** A standout feature is that agents might generate and run code (for example, running unit tests on generated code, or executing a data pipeline). To protect the system and host, all such execution should be sandboxed:
- We can use **WebAssembly (WASM)** sandboxing. Rust code (or generated code in some languages) can be compiled to WASM and executed in a limited environment (using engines like Wasmtime). WASM sandboxing ensures the code cannot access the host system arbitrarily – it will only have access to what we explicitly permit (e.g., no file system or network unless allowed). This provides a lightweight, in-process sandbox. The memory isolation of WASM and the ability to set execution time or memory limits is useful for containing potentially buggy or malicious code.
- Alternatively, use **Docker containers** as sandboxes. For instance, if agents produce a Python script or a binary program, the Testing agent could spin up a temporary Docker container that has no access to the host except a specific volume for code, run the program there, then destroy the container. This is heavier than WASM but allows testing in an environment closer to a real deployment (with certain libraries, etc.). Kubernetes can manage such job containers if running in a cluster.
- In either case, the orchestrator should mediate any code execution. An agent intending to run code would send a request (maybe via a tool-call interface) to something like an "execution manager" which does the sandboxing.
- **Secure Communication:** As mentioned, inter-agent comms on a network use TLS. We might generate a key/cert pair for the orchestrator and have agents trust the orchestrator's certificate (or share a cluster secret for symmetric encryption). This ensures that even if the agent swarm is distributed across machines or cloud nodes, eavesdropping on their message exchange is prevented.
- **Authorization and Limits:** Each agent should run with the principle of least privilege. For example, if an agent doesn't need internet access, we do not grant it. In Kubernetes, one can use network policies to restrict egress for certain agent pods. If an agent only needs to access an internal API (like the MCP server or a database), its container can be limited to only that endpoint. Within the Rust program, we ensure that an agent's capabilities are limited to what's defined by its role. (If using plugins/tools, the orchestrator should control which tools an agent is allowed to invoke.)
- **Auditing and Logging:** The system should log agent actions and communications for debugging and security auditing. Rust's single-binary nature makes it easier to have a unified logging system. The concept of **"Agent Provenance"** (tracking what each agent did) can be built in [3]. This means every change to the blackboard or any tool invocation can be recorded with the agent's identity and timestamp. Such logs both help in debugging the agent swarm's decisions and also serve as an audit trail in case something goes wrong or if we need to analyze how a solution was derived.

- **Fail-safes:** If an agent starts acting erratically (e.g., looping endlessly or producing nonsense due to an LLM hallucination), the orchestrator can terminate it and respawn a fresh instance. Timeouts should be set for agent tasks. Rust's concurrency allows us to use watchdog threads or async timeouts to ensure no agent hogs resources indefinitely. In an extreme case, the orchestrator can shut down the entire swarm if it detects a potential security issue or runaway condition, thereby containing any damage.

By leveraging Rust and careful architecture, the system will be **secure by default**. Rust's safety and the planned isolation (both in memory and in execution context) guard against many risks. In essence, even if one of the AI-driven agents tries something unintended, it's confined within a sandboxed, well-monitored environment – preventing the hypothetical "AI-pocalypse" by containing agents in controlled bounds [18].

Scalability and Deployment Modes

The tool is designed to run on a single machine or scale to a cluster:

- **Single-Machine Deployment:** The simplest mode is running the entire system on one computer (e.g., a developer's laptop or a server). In this case, the orchestrator and agents might all run within one OS process or a few processes. We can leverage **Rust's async and multithreading** for concurrency. For example, the orchestrator spawns each agent as a separate thread, or uses an async runtime (Tokio) to run each agent as a task. Because Rust has no GIL and efficient threading, multiple agents truly run in parallel on multiple cores [2]. This allows utilizing all CPU cores for different agents' workloads. Memory sharing is straightforward (in-process via pointers or channels). This mode is beneficial for low-latency communication between agents (no network overhead) and is easier to debug. The entire system compiles into a single binary which can simply be executed on the target machine – very low friction for deployment [3]. Users can configure the number of threads or agents according to their CPU/RAM limits.
- **Distributed (Cluster) Deployment:** For heavier tasks or cloud-based use, the system can be containerized. We can package the Rust binary (or binaries) into Docker images. There are a couple of ways to architect the cluster deployment:
- **All-in-One Service:** Run the orchestrator and agents within one container (essentially the single-process model above) and let that container scale vertically (more CPU). This is simple but doesn't fully utilize multiple machines.
- **Microservices Model:** Run the orchestrator as one service and each agent (or each type of agent) as separate services/containers. For example, you might have a **deployment** of `orchestrator-service`, and separate deployments for `planner-agent`, `coder-agent`, `tester-agent`, etc. The orchestrator, upon needing an agent, could send a request to the appropriate agent service (which could have multiple replicas behind a load balancer). This is more complex but allows independent scaling – e.g., if coding tasks are the bottleneck, run more instances of the CodingAgent service on the cluster. Communication in this model would be via HTTP/gRPC calls or message queues. Kubernetes can manage the scaling and service discovery between these components.
- **On-Demand Containers:** Leverage Kubernetes Jobs or temporary containers for agents. The orchestrator (running as a service) can use the Kubernetes API to spawn a Job or Deployment for an agent swarm when a task starts. Each agent comes up in its own container/pod, does its work, and terminates. This model truly treats agents as ephemeral. It adds some overhead (container startup

time), but for long-running tasks that might be acceptable, and it gives maximum isolation (each agent in its own pod) plus the ability to distribute across nodes.

In a cloud scenario, using Kubernetes offers benefits like automated restart of crashed pods, scaling based on load, and isolation of resources. For example, one could pin certain agents to GPU-equipped nodes (if one agent type needed a GPU for ML inference) while others run on CPU nodes.

- **State Management in Cluster:** When distributed, the blackboard can be a **shared data store** accessible to all agents. We could implement it as an in-memory data grid or simply use a lightweight database that all agents can reach (for instance, a Redis instance or a PostgreSQL, depending on the needed complexity). The choice might depend on the data size and access patterns: if the data is mostly text (prompts, code), a document store or even a git repository could hold it (though real-time coordination via git is tricky). A key-value store like Redis (with pub/sub) might be an elegant solution: agents post data to Redis and subscribe to channels for updates. Redis can be secured with passwords and network policies within the cluster.

- **Low Resource Environments:** The design remains usable on modest hardware. Because agents can be threads in one process, even a single-CPU machine can time-slice between agents using async (Tokio can schedule tasks cooperatively). If memory is limited, one can limit the number of agents or have them work more sequentially. The Rust binary can be compiled in release mode for efficiency and will have a relatively small memory footprint (especially compared to Python-based solutions, which might require tens or hundreds of MBs for the runtime overhead). We will also make components optional – e.g., if long-term memory (MCP integration) is not needed, it can be turned off to save resources (no need to run a separate memory server or store embeddings).

- **Extensibility and Maintainability:** By having a clear separation of components, the tool/repository will be maintainable and extensible. For instance, if in the future we want to add a new type of agent (say an “AI code reviewer” that adds comments to code), we can do so by implementing the Agent trait for that role and updating the orchestrator’s configuration. The use of Rust traits and generics can make adding new tools or agent capabilities straightforward, while keeping type safety. Rust’s powerful macro system could even be used to reduce boilerplate when defining new agent message types or logging routines [19].

- **Auditable Single Binary:** Deploying a Rust service is simplified by the fact that it can be a single binary with all functionality included [3]. This is advantageous for both cloud and edge deployments. On Kubernetes, the container image just needs that binary (no need for a heavy runtime). On a single machine, the user doesn’t have to install a runtime or dozens of Python packages – they just run the binary. This also reduces the attack surface and avoids dependency breakage issues. The binary can be scanned and audited for security since it’s self-contained and consistent across environments [3].

In essence, the system can start as a **single-machine application** for simplicity and then evolve into a **cloud-native distributed system** as needed, without major architectural changes. Thanks to Rust’s efficiency and cross-platform compilation, the same codebase can target both scenarios. Kubernetes and Docker are leveraged as needed to provide scaling and isolation, but are not mandatory for basic usage. This flexibility meets the requirement of running “on a single computer” and also being deployable to cloud clusters (with Docker/K8s as the backbone for multi-node deployments).

Integration of External Tools (MCP and Plugins)

To solve complex problems, agents often need to use external tools or services beyond their own logic. Our design incorporates a **tool interface** to allow this extensibility in a controlled way. Key points include:

- **Model Context Protocol (MCP) Integration:** As mentioned, MCP is an emerging standard for connecting AI agents to external data and actions. By implementing an MCP client within our framework, we immediately gain access to a wide range of existing tools and extensions that speak MCP [20] [15]. For example, an MCP server might provide long-term memory, as well as other capabilities like retrieving emails, querying a knowledge base, or even controlling a web browser. Rather than reinvent each integration, our agents can invoke MCP calls. In practice, this means if an agent's task says "search the knowledge base about X", the orchestrator/agent translates that into an MCP call (which might be an HTTP request to an MCP server listening on a port). The result comes back (e.g. relevant data) and is then placed on the blackboard for agents to use. MCP is **secure and scalable**, and saves us from writing custom adapters for each external API [5]. We just need to follow the MCP protocol format and use available SDKs or direct HTTP calls. The user can configure the addresses of MCP servers (for memory, for tools, etc.) in a config file.
- **Custom Tools and APIs:** Not everything will have an MCP server (especially user-specific internal tools). So, the framework will also allow registering custom tool handlers. We can define a trait like `Tool` with a method `execute(params) -> Result` and allow agents to call `Tool` by name. For instance, if a user wants the agents to be able to **run database queries**, they might implement a Tool that connects to their database. The agent (likely via the LLM's reasoning or a hard-coded call) would invoke `Tool("database_query", "SELECT ...")`, and the orchestrator would route this to the actual tool implementation, then return the result to the agent or blackboard. This resembles the *function calling* mechanism in frameworks like OpenAI's function API, but implemented in Rust. In fact, if using an LLM, the agent's prompt can list available tools (including ones backed by MCP or custom Rust code), and the LLM's output can indicate which tool to use. The orchestrator then interprets that and calls the appropriate function in Rust. This design was inspired by approaches in other agent frameworks which let LLMs choose functions instead of hardcoding rules [21] [22]. The difference here is we ensure these calls are executed by safe Rust code.
- **Examples of Tool Extensions:** Besides memory, we could have tools for:
- *Code Compilation/Execution:* A tool that compiles a given code snippet and returns any errors (this could simply invoke `rustc` or `gcc` on sandboxed code, or use an online compiler API).
- *Web Requests:* If the agent needs to fetch data from the web (for example, to get documentation or verify something online), a tool could be provided (with caution) to perform HTTP GET requests. This would be tightly controlled (maybe only allowed to certain domains) for security.
- *Data Pipeline Operations:* For tasks involving data processing, tools might interface with data sources or processing libraries. For instance, an agent might call a "run Spark job" tool or a "query CSV dataset" tool. The user can integrate their data processing engines through such hooks.

- *DevOps/Cloud Tools:* If the problem involves deploying the code, an agent could use a tool to interact with Docker or Kubernetes (e.g., building an image, deploying a service). These would execute real commands but under supervision.

- **Configurable and Sandbox Tools:** All tools can be enabled/disabled via configuration. If a user doesn't want agents to have internet access, they simply don't provide the WebRequest tool or don't configure an MCP server that can browse. This way, the "swarm" the user spins up is tailored to the problem domain and security context they want. Tools themselves should be designed securely. For example, if providing a shell command tool, it should sanitize inputs or run the command in a safe environment to avoid injection issues.

- **Logging and Monitoring of Tool Use:** The orchestrator will log every tool invocation (which agent called what tool with what parameters). This transparency is important, especially if agents are autonomous – the user needs to know what actions were taken on their behalf. MCP calls, for instance, are essentially tool calls that will be logged as well (MCP's design itself emphasizes that the AI app decides when to call a tool, often after the LLM suggests it [23] [24]). We'll make those steps visible in logs or UI.

By designing around a **pluggable tool interface**, we make the agent swarm highly **extensible**. Users or developers can continuously augment the system with new capabilities as new needs arise. Want agents to produce a UML diagram of the planned system? Add a "DiagramTool" that can render diagrams from a description. Because our framework is in Rust, many existing libraries (for database access, HTTP, etc.) can be integrated as tools. And the use of MCP where possible means we align with an ecosystem of tools that are already being built for AI agents, future-proofing the design [25] [26] .

Example Workflow: Building an API Service

To illustrate how all these pieces come together, consider a user asks the agent swarm to **"Build a REST API for a TODO list application"**. Below is a possible workflow with our design:

1. **User Input & Orchestrator Kickoff:** The user provides the problem description to the orchestrator (perhaps via a CLI or web UI that wraps the orchestrator). The orchestrator creates a new **blackboard** for this task and records the problem details. It then spawns a **Planner Agent** to strategize.
2. **Planning Agent Activity:** The Planner Agent reads the problem from the blackboard. Using an LLM (with a system prompt like *"You are a software architect"*), it generates a plan: e.g. *"We need: (a) Data Model for TODO items, (b) API Endpoint for creating a TODO, (c) Endpoint for listing todos, (d) Endpoint for marking complete, (e) basic documentation."* This plan (a list of tasks or modules) is written to the blackboard [7] . The planner might also suggest which agent type should do each (marking some for "Coder", one for "DocAgent", etc.).
3. **Orchestrator Spawns Coding Agents:** Seeing the plan, the orchestrator starts (or awakens) multiple **Coding Agents** – e.g., one for the Data Model and one for each API endpoint. It provides each agent with the relevant task description (from the plan) either via a targeted message or by each agent picking an unclaimed task from the blackboard. For instance, CodingAgent1 is assigned "Implement the data model and database integration", CodingAgent2 gets "Implement create_todo endpoint", etc. All coding agents have access to the overall plan on the blackboard, which gives context.
4. **Coding Agents Generate Code:** Each Coding Agent uses its logic to produce code for its assignment. Suppose we integrate OpenAI's API or another LLM: the agent's code could involve prompting the LLM with the task (and any relevant schema or language requirements). The LLM's output (the code) is then possibly reviewed by the agent (to fix small errors) and posted to the blackboard as a code snippet. In practice, we might have the agent directly commit to a git repository and then note the commit hash on the blackboard. For simplicity, imagine each agent writes a chunk of code to the shared memory with a label (e.g., "data_model.rs", "api_create.rs").

5. **Parallel Development:** The agents work in parallel (thanks to our multi-threaded setup). While the Data Model agent is designing the database schema, the Endpoint agents start writing controller logic. They might all need the definition of the data model. In our design, the Data Model agent can publish an interim interface spec on the blackboard (like a struct definition for a TODO item) once ready, so that others can use it. Because the blackboard is globally visible, the CreateEndpoint agent sees that and incorporates it. In case a coding agent needs something not yet present (say endpoints need the data model which is still being coded), agents could either wait or the orchestrator can orchestrate the order (e.g., start Data Model agent slightly before others, or have them communicate if blocking).
6. **Testing Agent Validation:** As soon as any code is posted, the **Testing Agent** kicks in. The Testing Agent retrieves the current code (all components) from the blackboard. It might generate some test cases automatically (possibly via an LLM prompt like "Write unit tests for the API") or use predefined tests if provided. Then it uses the sandbox execution module to run the tests. For example, it compiles the code inside a WASM runtime or docker container. Suppose the test for "create_todo" fails because of a bug. The Testing Agent posts the test results (with an error trace) onto the blackboard.
7. **Feedback Loop:** The CodingAgent responsible for that part sees the test failure on the blackboard. It then goes back to fix the code. This may involve calling the LLM with the error message to suggest a fix, or the agent's own logic adjusting the code. The updated code is posted again. This iterative loop continues until tests pass for all features. Throughout this, all agents see the evolving state: for instance, another agent (say a **Reviewer Agent** if present) could also comment on code quality. If we had a "Critic" agent, it might read the code and point out potential improvements, writing those suggestions on the board for coding agents to consider (this is similar to an agent debating or refining on the blackboard, which is supported by the blackboard approach [13]).
8. **Documentation Agent Finalizes Docs:** Meanwhile, a **Documentation Agent** has been waiting for the overall implementation to stabilize. Once the core code is in place (or as it's being produced), the Doc Agent reads the code from the blackboard and generates documentation. For example, it might produce a README with instructions on how to use the API, or inline doc comments for each endpoint function. It uses an LLM tuned for summarizing code or a template-based approach to format the documentation. The output is placed on the blackboard (e.g., "API_DOC.md" content).
9. **Completion and Tear-down:** When the orchestrator observes that all tasks are completed (all code sections done, tests passing, docs written), it triggers a final step. It might have a **Build Agent** do one final integration (like making sure all modules compile together, packaging the project). The orchestrator then gathers the final artifacts: the source code files, documentation, maybe a compiled binary of the service. It presents these to the user (perhaps zipping them up or pushing to a repository). Finally, the orchestrator signals all agents to terminate. They shut down gracefully, freeing memory. The blackboard for that project can be archived or cleared. The entire swarm that was launched for the problem now spins down, fulfilling the requirement that agents live only for the duration of the problem.

Throughout this process, if we had integrated an external **MCP memory**, say the user had relevant past notes about TODO apps, an agent could have queried that at the start. For instance, the Planner Agent might call a memory tool: *"Recall if the user had any preferences for this API"*. The MCP server could return a snippet from last week's conversation about using PostgreSQL, which the planner would then include in the plan (ensuring the data model uses Postgres). This would have been seamlessly done via our tool interface to MCP.

This example demonstrates how multiple agents with different roles cooperate: **planning, coding, testing, documenting** – all in parallel and orchestrated via a shared memory and central coordinator. Problems are broken down and solved faster than a single agent could do sequentially, and the design ensures they can run on one machine or many. Each agent had some short-term memory (their current subtask and any intermediate LLM thoughts) and shared everything useful on the board for others. The result is a fully implemented piece of software (the API) along with tests and docs, achieved by an ephemeral swarm of agents that was spun up on demand and then spun down.

Conclusion

We have outlined a comprehensive design for a Rust-based tool that enables spinning up a **swarm of intelligent agents** to collaboratively solve programming tasks (and extendable to many other tasks). The design emphasizes:

- **Robust architecture:** A central orchestrator with a shared blackboard for coordination, enabling agents with specialized roles to work together efficiently [7].
- **Secure, efficient implementation:** Rust provides speed and memory safety, important for running untrusted code and multi-threaded workloads [1]. The use of sandboxing (WASM/containers) and encrypted communication further ensures security.
- **Memory and communication model:** A hybrid memory approach with global shared context and optional private state meets the needs of collaboration without losing individual agent utility [13] [8]. Communication is handled through structured message passing and shared state, allowing for parallelism and clear information flow.
- **Scalability:** The solution scales from a single laptop to a Kubernetes cluster. It can exploit multi-core CPUs on one machine, and multiple nodes in cloud, orchestrated via Docker/K8s. The Rust “single binary” deployment model eases this portability [3].
- **Extensibility with tools:** By integrating with the Model Context Protocol and providing a plugin interface, the agent swarm can use external tools and long-term memory stores on demand [4] [5]. This makes the system adaptable to various domains (coding, data processing, etc.) simply by configuring different tools or agents.
- **Parallel problem solving:** The framework inherently supports dividing work among agents (e.g. different coding tasks, or concurrently handling parts of a data pipeline). This yields efficient problem solving, as seen in the example where multiple parts of the project were built simultaneously.

In summary, this design leverages the best of modern multi-agent system ideas (like blackboard architectures and function-driven AI agents) implemented with Rust’s strengths (performance, safety, concurrency). The result is a **flexible, secure, and powerful agent-swarm framework**. Users can instantiate a swarm tailored to their problem, confident that the agents will communicate and cooperate effectively, all while running within resource-friendly bounds on their hardware. By following this blueprint, one could proceed to implement the system as an open-source Rust repository, enabling developers to solve complex tasks through dynamic swarms of specialized AI agents.

Sources:

1. Ian Rumac, *"AI Agents: Building AI Primitives with Rust"* – discussing Rust’s safety and concurrency benefits for AI agents [17] [19].

2. Zectonal (2024), *"Why We Built Our AI Agentic Framework in Rust from the Ground Up"* – notes on spawning agents on-demand in Rust and single-binary deployment [27] [3].
3. Ellie Zubrowski (2025), *"What the heck is Model Context Protocol (MCP)?"* – describes MCP as an open standard for connecting AI to external tools/data [4] [5].
4. Han et al. (2025), *"Exploring Advanced LLM Multi-Agent Systems Based on Blackboard Architecture"* – introduces the blackboard architecture for multi-agent collaboration [7] [13].
5. Aman Raghuvanshi (2025), *"Agentic AI: Multi-Agent Architectures Explained"* – outlines core MAS components and memory models [8] [12].

---

[1] [17] [18] [19] AI Agents: Building AI Primitives with Rust | Shuttle
https://www.shuttle.dev/blog/2024/04/30/building-ai-agents-rust

[2] [3] [21] [22] [27] Why We Built Our AI Agentic Framework in Rust From the Ground Up | by Zectonal | Medium
https://zectonal.medium.com/why-we-built-our-ai-agentic-framework-in-rust-from-the-ground-up-9a3076af8278

[4] [5] [14] [15] [20] [25] [26] What the heck is Model Context Protocol (MCP)? And why is everybody talking about it?
https://pieces.app/blog/mcp

[6] [8] [12] Agentic AI #6 — Multi-Agent Architectures Explained: How AI Agents Collaborate | by Aman Raghuvanshi | Jul, 2025 | Medium
https://medium.com/@iamanraghuvanshi/agentic-ai-7-multi-agent-architectures-explained-how-ai-agents-collaborate-141c23e9117f

[7] [9] [10] [11] [13] [16] Exploring Advanced LLM Multi-Agent Systems Based on Blackboard Architecture
https://arxiv.org/html/2507.01701v1

[23] [24] Introducing Pieces MCP Server: Add Long-Term Memory to your favorite AI tools
https://pieces.app/blog/introducing-the-pieces-mcp-server