- Error isolation — limiting the propagation of hallucinations across modules.
- Reusability — allowing subtasks to be reused across different project types.

This design is consistent with frameworks such as *Tree-of-Code* [13], which emphasizes modular decomposition for robustness in complex codebases.

Role-Specialized Agents

To execute subtasks, CodeCodez employs role-specialized LLM agents, each fine-tuned (or prompted) for domain expertise. Drawing from *Mixture-of-Agents* [4], *AgentCoder* [15], and *AdaCoder* [16], the system integrates multiple agents into a collaborative ecosystem:

- **System Architect Agent** → responsible for high-level project scaffolding, guided by hierarchical orchestration strategies [3].
- **Frontend Agent** → specialized in UI frameworks and design patterns.
- **Backend Agent** → focused on API development, database integration, and business logic.
- **Testing Agent** → designs and executes test cases, aligned with iterative testing loops as in [15].
- **Documentation Agent** → generates structured, human-readable project documentation, ensuring transparency**.**

To improve efficiency, CodeCodez leverages dynamic pruning strategies similar to *AgentDropout* [8], ensuring that redundant agents are removed and token usage remains cost-efficient. Furthermore, context engineering methods [17] are employed to maintain task-relevant prompts and avoid context bloat, enabling each agent to focus on its domain without distraction.

Iterative Validation and Self-Reflection

CodeCodez integrates an iterative validation loop to enhance reliability. Following the principles of *AgentCoder* [15], *CodeCoR* [14], and *AdaCoder* [16], the framework enforces a generate → test → refine cycle:

1. **Code generation** — initial output by domain agents.
2. **Testing and debugging** — validation by the Testing Agent, which executes unit and integration tests.