Harnessing the Power of Large Language Model-Based Agents in Software Engineering

Sajal Agarwal
5 min read · Just now

In recent years, the leap in artificial intelligence, especially in large language models (LLMs), has unlocked new frontiers for software engineering (SE). LLM-based agents are transforming the way developers handle complex tasks, ranging from requirements engineering to debugging and testing. This survey-based article explores the capabilities of LLM-based agents, their architecture, applications across SE stages, and future opportunities. Dive into the paradigm shift, as we see how AI is not just assisting but increasingly driving software development from concept to completion.

Introduction to LLM-Based Agents in Software Engineering

LLMs like GPT-3, BERT, and others have proven adept at generating human-like responses and solving a variety of tasks. However, standalone models face limitations, especially in dynamic, multi-step tasks in SE. LLM-based agents, on the other hand, enhance these models by allowing them to interact with external tools, receive and process feedback, and even collaborate with other agents and humans. With these additional layers of functionality, LLM-based agents can now perform iterative tasks that require adaptability, such as code generation, requirements validation, and testing.

Core Components of LLM-Based Agents

At the heart of LLM-based agents are four essential components:

- **Planning:** This component breaks down complex tasks into manageable sub-tasks, allowing the agent to follow a structured approach. Planning is crucial for

code generation, where an LLM-based agent may need to create sequential functions or modular code segments.

- **Memory**: Agents use memory to store historical interactions, thoughts, and actions. This allows them to learn from previous experiences and enhance efficiency. Memory is especially useful in debugging, where agents need to refer back to past errors or tests.
- **Perception**: Perception allows agents to gather information from various sources, including textual, visual, or even auditory inputs. For example, in requirements engineering, an agent might analyze natural language input from stakeholders.
- **Action**: Finally, the action component enables agents to interact with their environment, leveraging tools or accessing resources to complete their tasks. For instance, an LLM-based agent could use a code compilation tool to check the correctness of generated code

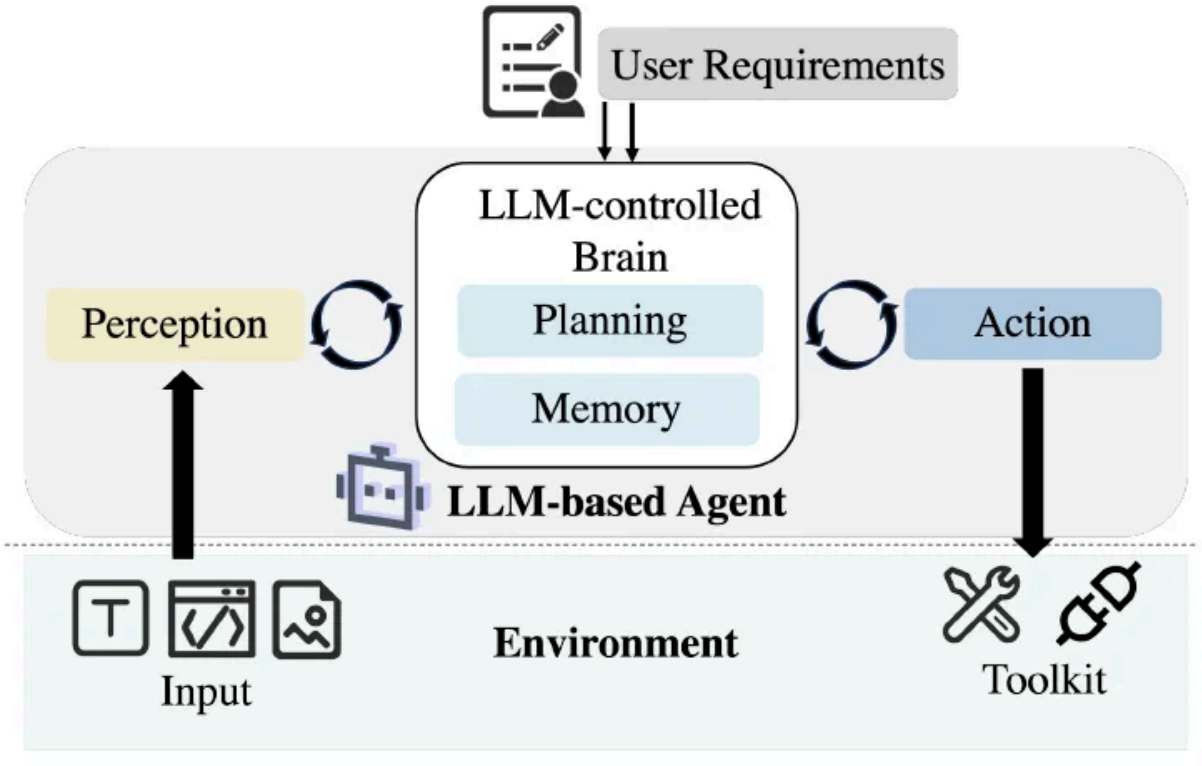


Diagram showing the architecture of an LLM-based agent with the four components (Planning, Memory, Perception, and Action)

Applications Across the Software Development Lifecycle

Requirements Engineering (RE)

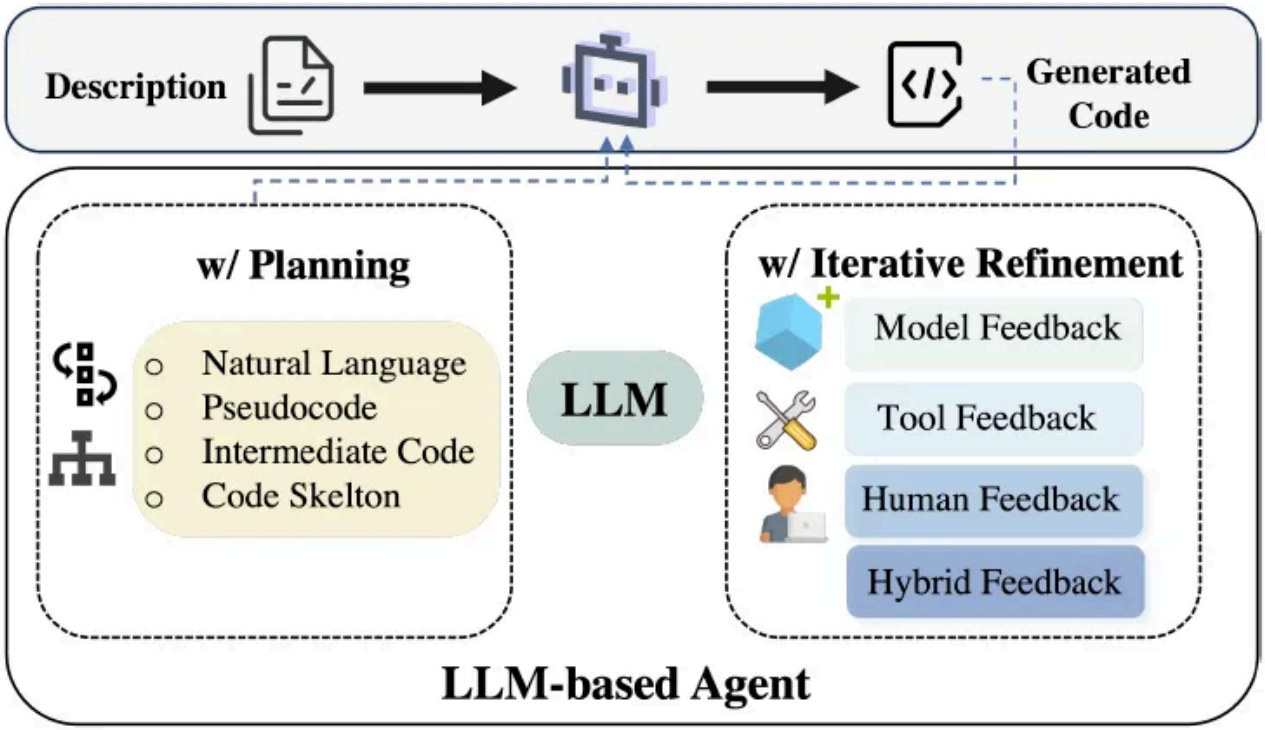
LLM-based agents in RE can automate several phases: eliciting requirements from stakeholders, modeling requirements through diagrams, negotiating conflicting needs, and verifying requirements. Multi-agent systems like **Elicitron** and **SpecGen** have emerged as pioneers, setting agents up to play various roles — stakeholders, engineers, and documenters. This setup allows agents to simulate real-world interactions, helping capture a more accurate picture of project requirements.

Agents	Multi-Agent	Covered RE Phases					
		Elicitation	Modeling	Negotiation	Specification	Verification	Evolution
Elicitron [54]	✓	✓					
SpecGen [55]	×				✓		
Arora *et al.* [56]	✓	✓		✓	✓	✓	
MARE [57]	✓	✓	✓		✓	✓	

Existing LLM-based Agents for Requirements Engineering

Code Generation

While standalone LLMs can generate code snippets, agents bring this process to the next level by breaking down code generation into sub-tasks. Agents like **CodePlan** and **AgentCoder** use planning to create structured, modular code. Feedback loops with human or automated tool inputs ensure accuracy and prevent common pitfalls like syntax errors or incorrect logic.



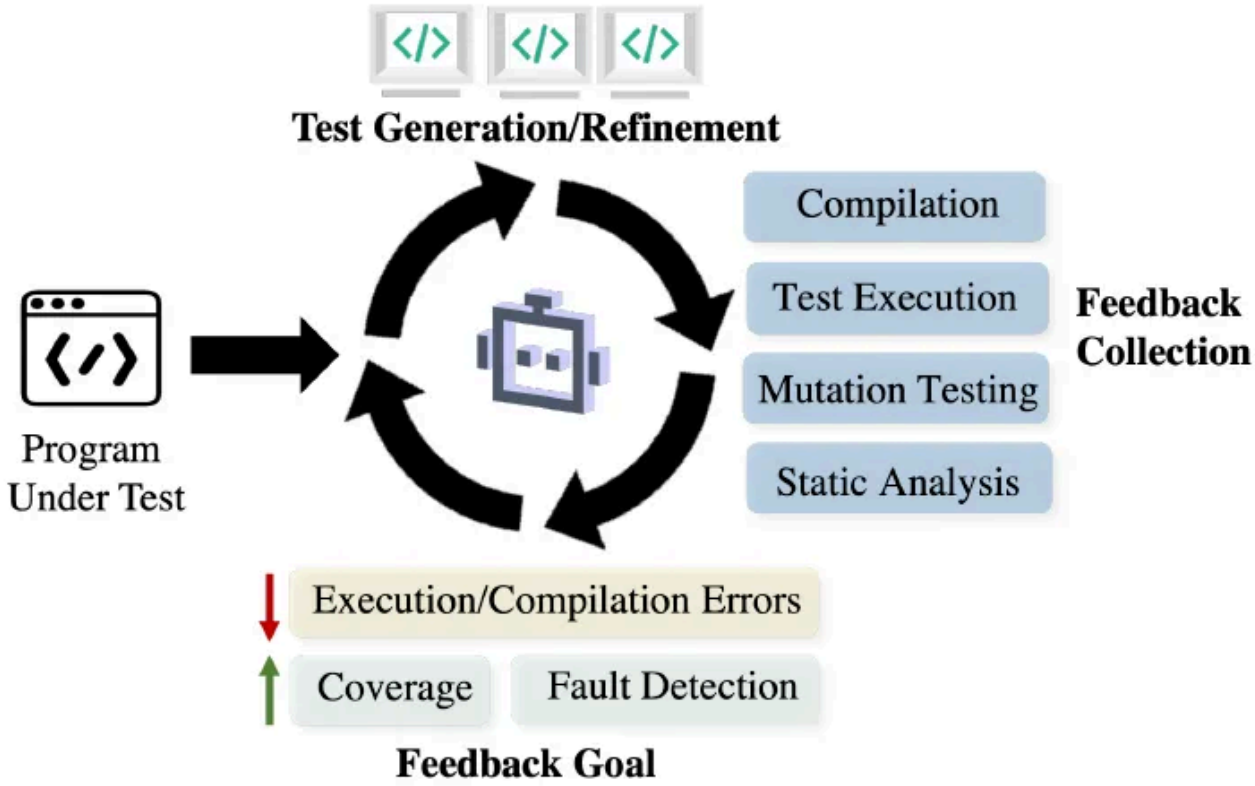
Pipeline of LLM-based Agents for Code Generation

Static Code Checking

Static code checking involves analyzing code for errors without executing it. Multi-agent systems, such as **ICAA** and **GPTLENS**, mimic human peer review, with one agent proposing solutions and another acting as a critic. This multi-role approach has led to impressive results, enabling agents to identify vulnerabilities in code, such as memory leaks or incorrect API usage, at levels comparable to, or even surpassing, traditional static analysis tools.

Testing and Debugging

Testing has seen tremendous growth with LLM-based agents generating test cases, identifying bugs, and even repairing code. Agents like **TELPA** increase test coverage by generating tests focused on hard-to-reach code branches. **MuTAP** uses mutation testing to generate robust unit tests that can reveal hidden bugs. Debugging agents like **AgentFL** and **AUTOFL** aid in localizing and repairing bugs, using tools to analyze logs, feedback, and historical error data.

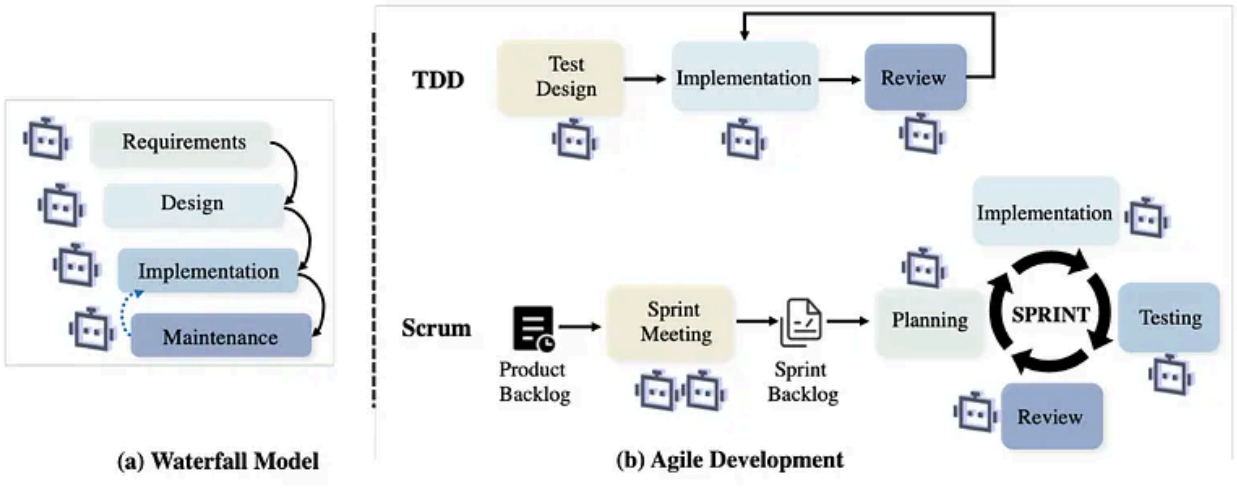


Pipeline of LLM-based Agents for Unit Testing

End-to-End Software Development

End-to-end software development is where LLM-based agents shine. By emulating a full development lifecycle, agents can handle a project from requirements to deployment. Systems like **ChatDev** and **Self-Collaboration** assign each agent a unique role, much like a team of developers, testers, and project managers. These

agents use predefined process models (waterfall or agile) to create robust, deployable software, all while integrating real-time feedback and collaboration.



Process Models Adopted by LLM-based Agents for End-to-end Software Development9

4. Key Metrics and Evaluation

Evaluating the effectiveness of LLM-based agents involves several metrics, primarily:

- **Code Quality**: Measuring correctness, readability, and maintainability.
- **Error Rate**: Quantifying how often generated code contains errors.
- **Coverage**: In testing, this is crucial for ensuring all parts of the codebase are tested.
- **Efficiency**: Metrics include the time taken to complete a task and the number of iterations required.

In an ablation study by ChatDev, removing the memory component increased error rates by 20%, showing how essential this component is for long-term, iterative tasks.

5. Future Opportunities and Challenges

Despite their promise, LLM-based agents face several challenges, including:

- **Human-Agent Collaboration**: Ensuring seamless, understandable interactions between human developers and agents is crucial.
- **Memory Management**: Storing and retrieving relevant information from memory without overwhelming the system remains a key research area.

- **Ethical and Bias Considerations:** Agents may unintentionally introduce biases or overlook certain vulnerabilities, requiring robust frameworks for fairness and ethics.

6. Conclusion

LLM-based agents are redefining the boundaries of what AI can achieve in software engineering. By equipping language models with memory, planning, perception, and action capabilities, researchers have enabled these agents to tackle complex SE tasks with remarkable autonomy and efficiency. From generating code to debugging and testing, LLM-based agents are not just tools — they are becoming active collaborators in the software development process. As we continue to refine these systems, LLM-based agents could soon become indispensable in the development toolkit.

Edit profile

Written by Sajal Agarwal

1 Follower

More from Sajal Agarwal