Autograder in computing education with multi agent frameworks

Introduction

The increasing enrollment in Computer Science courses and the demand for effective programming education have highlighted the critical role of assessment and feedback [1]. Auto-graders have emerged as a significant innovation in this space, offering the potential for efficient and scalable evaluation of student code [2]. However, the adoption of auto-grading has presented several challenges, impacting both instructors and students [4]. This literature review examines these challenges and explores the potential of multi-agent frameworks, leveraging large language models, to offer more sophisticated and effective solutions.

Literature databases and key words used

We employed a systematic literature review strategy to identify relevant studies on LLM-Based Multi-Agent systems in software engineering. Our strategy began with a keyword-based search on Carnegie Mellon's database eresources and tools, which exposed us to many industry databases like ACM Digital Library and IEEE Xplore. We utilized two sets of keywords: one focused on "agent words" such as "Agent", "LLM", and "Large Language Model", and the other on "SE words" specific to requirements engineering, code generation, quality assurance, and software maintenance, combining these with the AND operator for each SDLC phase. To broaden our search to autograder systems, we additionally included keywords such as "Autograder", "Automated Grading", "Code Assessment", "Automatic Code Evaluation", "Program Synthesis Grading", "AI Grader", "LLM-based Autograder", "Automated Feedback", "Code Review Automation", and "Educational Code Assessment". This comprehensive keyword strategy ensured the inclusion of studies at the intersection of LLM-based multi-agent systems, software engineering, and automated grading. Subsequently, we applied inclusion and exclusion criteria in a three-phase process to filter out irrelevant papers based on factors like duplication, publication type, lack of focus on publication date eg: before November 2022, lack of relevance to software engineering and absence of experimental results . To further broaden our search, we conducted both backward and forward snowballing on the initially identified relevant papers until no new studies were discovered. This comprehensive approach aimed to provide a thorough mapping of the current research landscape of LMA applications in software engineering. Additionally, to capture the most recent advances and grey literature, we leveraged Perplexity's AI powered search with prompts such as "What are the latest research papers and preprints on LLM-based multi-agent systems in software engineering?", "Survey new frameworks for LLM-powered autograder agents," and "Identify recent experimental results for multi-agent LLMs in requirements engineering and software maintenance." This approach ensured our review included both peer-reviewed and emerging industry trends.

Challenges of Traditional Autograding in Computing Education

Instructors have reported a trade-off between the quick, binary feedback provided by many auto-graders and the more nuanced, formative feedback achievable through manual grading [4]. Primarily relying on static and functional analysis, existing auto-graders often struggle to identify crucial learning aspects such as anti-patterns and stylistic or idiomatic errors [4]. While improvements are being made in this area [4], the fundamental limitation of providing in-depth conceptual guidance remains a concern [6].

Furthermore, the ecosystem of available auto-grading tools is fractured, with features and even entire platforms being abandoned over time, leading to frustration and hindering consistent adoption [4]. Instructors value automated feedback, customizability, and broad language and framework support in auto-graders [9]. However, they encounter challenges when tools lack the flexibility to incorporate their preferred grading strategies, specify course policies, or provide meaningful feedback for issues beyond basic correctness, such as memory leaks or inefficient algorithms [9]. The complexity of setting up and maintaining custom auto-grading solutions also poses a barrier to wider adoption by instructors [7].

The Potential of Multi-Agent Systems Leveraging Large Language Models

Recent advancements in the development of large language models have opened new avenues for creating more intelligent and versatile educational tools [13]. Multi-agent systems (MAS), which involve the collaboration of multiple autonomous agents to solve complex tasks [14], offer a promising framework for addressing the limitations of traditional auto-graders.

Multi-Agent Systems in Software Engineering

The application of MAS is gaining traction in various software engineering tasks throughout the software development lifecycle (SDLC) [15]. These applications include:

- **Requirements Engineering**: Multi-agent frameworks can assist in generating, evaluating, and prioritizing user stories through collaborative processes [20].
- **Code Generation**: MAS can automate coding tasks by leveraging role specialization and iterative feedback loops among agents with roles like Orchestrator, Programmer, Reviewer, and Tester [21].
- **Quality Assurance**: LMA systems are being employed to enhance testing, vulnerability detection, bug detection, and fault localization processes [22].
    - **Testing**: Frameworks like Fuzz4All utilize LLMs to generate diverse test inputs across multiple programming languages [22]. Multi-agent systems can also generate and refine unit tests iteratively to improve code coverage [24].
    - **Vulnerability Detection**: Systems like GPTLens employ adversarial LLM agents (Auditor and Critic) for progressive vulnerability detection in smart contracts [22]. MuCoLD uses multi-role consensus through LLM discussions to improve vulnerability detection by incorporating diverse perspectives [22].
    - **Bug Detection**: The Intelligent Code Analysis Agent (ICAA) leverages LLMs and various tools for static code analysis to detect code errors and functional logic inconsistencies [22].

    - **Fault Localization**: AGENTFL, a multi-agent system, simulates human debugging behavior to localize bugs within large codebases through stages like comprehension, navigation, and confirmation [16]. RCAgent utilizes LLM-based agents to perform root cause analysis in cloud environments [22].
- **Software Maintenance**: MAS are being used for debugging by employing specialized agents for bug reproduction, fault localization, patch generation, and validation [35]. They also aid in code review by identifying bugs, code smells, and suggesting optimizations [36].

Architectural Patterns in Multi Agent Systems

Multi agent systems consist of multiple autonomous agents working together or in parallel to solve problems that often exceed the capability of any single agent. The following surveys key architectural patterns in such systems, describing their purpose, implementation strategies, real-world examples, code mechanics and relevant references. A multi-agent design pattern can be understood as a structured interaction protocol between agents, essentially, it defines how agents communicate and coordinate to solve tasks . By leveraging these patterns, we can assemble a "dream team" of specialized agents collaborating toward a shared goal, tackling complex challenges too big for one agent . The patterns covered here include Concurrent Agents, Sequential Workflow, Group Chat, Handoffs, Mixture of Agents and Multi-Agent Debate, each discussed in detail below.

Concurrent Agents

Description: *Concurrent Agents* is a pattern where multiple agents operate in parallel, either on independent tasks or on the same task, to increase throughput or bring diverse perspectives. Instead of a strict turn-taking or pipeline, concurrency allows agents to work simultaneously. For example, a single input can be broadcast to multiple agents that process it concurrently. This pattern is useful for speeding up workflows or for assembling different skills/approaches to a problem. It embraces an event-driven or publish subscribe style communication where all agents subscribed to a topic receive the message at once.

Implementation Strategies and Tools: Common implementations use asynchronous messaging or multi-threading to run agents in parallel. An event bus or topic-based routing is often used: one agent publishes a message that multiple agents subscribe to, triggering parallel handling . Frameworks like Microsoft's AutoGen provide a pub-sub mechanism for this, agents can subscribe to a default topic so that any new task broadcast is picked up by all concurrently . In distributed settings, streaming platforms (Kafka, Pulsar) can serve as the backbone for concurrency, treating agents as consumers of topics. The actor model naturally supports concurrent agents each running in its own process/thread. Key considerations are managing shared state and combining results. Often a coordinator or aggregator might collect outputs from concurrent agents or the agents might publish their results to another topic for further processing.

Code Example: Below is a pseudocode example of broadcasting a task to two agents concurrently and waiting for both results. This could be implemented with asyncio, threading, or an event bus. Each agent logs its progress in parallel, simulating the behavior shown in AutoGen's example:

```
# Pseudocode for concurrent agent execution
def agent_process(name, task):
    print(f"{name} received {task}")
    time.sleep(2)
    result = f"{name} result for {task}"
    print(f"{name} completed {task}")
    return result

task = "Analyze Report"

agents = ["Agent A", "Agent B"]
futures = []
for agent_name in agents:
    futures.append(async_execute(agent_process, agent_name, task))

results = [future.result() for future in futures]
print("All agents done. Results:", results)
```

In this pattern, all agents operate simultaneously upon receiving the task. The system must then decide how to handle multiple results, e.g: aggregating them, choosing the fastest response or presenting all outcomes to a human. The *Concurrent Agents* pattern thus emphasizes parallelism and independent processing, enabled by message routing infrastructure that delivers the same message to all interested agents.

References: Multi-agent concurrency is discussed in AutoGen's documentation, demonstrating how multiple processor agents subscribe to a topic to handle tasks in parallel. Event-driven agent architectures for concurrency are further elaborated by Confluent, which outlines patterns translating to data streaming contexts . These concurrent patterns relate to the broader concept of *blackboard systems* (all agents reading the same information) and *publisher subscriber models* in distributed systems.

Sequential Workflow

Description: *Sequential Workflow* is a pattern where a task is handled through a fixed sequence of agents, each performing a specific sub task in turn. The agents form a pipeline: the first agent processes the initial input and passes its output to the next agent and so on until the final result is produced. This deterministic chaining is useful when a problem can be decomposed into discrete stages, each handled by specialized expertise. The goal is a well-defined, reproducible process where each agent's role is predetermined, ensuring clarity of responsibilities in the multi-agent team.

Implementation Strategies and Tools: The simplest implementation is a static orchestration that invokes Agent1 then Agent2, etc passing outputs along. For example, one can use a series of API calls or function calls, each representing an agent's operation. In more decoupled systems, a topic per agent approach can

be used: Agent1 publishes to a topic that Agent2 subscribes to, and so forth . AutoGen's core supports this via type based subscriptions, each agent listens on a topic named after its role, and the previous agent publishes a message of that type . Orchestration frameworks like LangChain often implement sequential chains where the output of one tool or agent feeds the next. Business Process Modeling tools or workflow libraries (e.g. Apache Airflow, Prefect) can also coordinate sequential tasks. The key is defining the ordering and data passing. In LLM based systems, an orchestrator agent might maintain the flow, e.g: first ask a planning agent for a plan, then invoke a solving agent, then a verification agent, sequentially.

Code Example: Below is pseudocode showing a simple sequential workflow. Three agents (A, B, C) operate in a fixed order, passing along a message. Each agent's process() function transforms the input and returns an output for the next agent:

```
# Pseudocode for sequential workflow of agents A -> B -> C
message = initial_input
message = AgentA.process(message)   # Agent A handles first step
message = AgentB.process(message)   # Agent B handles next step
message = AgentC.process(message)   # Agent C handles final step
print("Final result:", message)
```

This could represent, for example, AgentA = "Parser", AgentB = "Reasoner", AgentC = "Responder". In a more explicit pub-sub implementation, AgentA might publish message to a topic that AgentB subscribes to, etc., but the end effect is the same: a pipeline of transformations. Deterministic ordering is maintained either by code structure or by designating a chain of topics. The sequential pattern ensures each subtask is completed before the next begins, which is useful for **well-defined task pipelines** and makes debugging/tracing easier (at the cost of potential slowdowns if tasks could be parallelized).

References: The concept of sequential multi agent workflows is exemplified by AutoGen's marketing copy pipeline. The approach of using an LLM as a central orchestrator for sequential tasks is detailed in the HuggingGPT paper , where the LLM essentially guides a step-by-step execution of various AI models. These illustrate the pattern's usefulness in structured task decomposition.

Group Chat

Description: *Group Chat* is a pattern where multiple agents (and potentially humans) participate in a shared conversation thread, collaborating by turn-taking in a multi-turn dialogue. All agents listen to the same conversation context (like being in the same chat room) and contribute messages one after another . Unlike a strict sequential pipeline, a group chat is often interactive and dynamic, agents might respond to each other or to a human user, and the sequence of turns can evolve based on the discussion. Typically, each agent has a distinct role or specialization, and the conversation allows them to collectively solve a problem or generate content. Under the hood, there is usually a mechanism to decide which agent speaks when, to avoid all agents talking at once.

Implementation Strategies & Tools: Implementation often relies on a conversation manager or a simple policy to alternate turns. One approach is a round-robin scheduler: e.g. always loop through agents in a fixed order (ensuring one message at a time). A more flexible approach is having a special agent (or function) act as a *moderator* or *group chat manager* that, upon each new message, decides who should reply next . This decision can be hardcoded (like alternate A, B, A, B for two agents) or computed by an LLM itself (e.g. feeding the dialogue so far into an LLM prompt that picks the most appropriate next speaker based on context). In Microsoft's AutoGen, for example, a GroupChatManager agent monitors the conversation and uses either a round-robin or an LLM-based function to choose the next agent to speak . All agents subscribe to a common topic representing the chat, so when one agent publishes a message, the others (and the manager) receive it . From a tooling perspective, frameworks that support multi-party dialogues or role-play (like some chatbot platforms) can be adapted for this. Even without a special framework, one can implement a loop in code that feeds the current conversation history plus a role prompt into each agent model in turn. It's important to maintain a shared memory of the conversation (the transcript) accessible by all agents, so they have context when it's their turn.

Code Example: The pseudocode below illustrates a simple moderated group chat. We have a set of agents and a conversation loop. A special manager decides who speaks next (here we use a round-robin for simplicity). Each agent, when called upon, generates a message based on the shared conversation history, which is then broadcast to all participants:

```
agents = [AgentRole1(), AgentRole2(), AgentRole3()]  # e.g., Writer,
Illustrator, Editor
manager = ConversationManager(agents)
conversation = []  # shared message history


conversation.append(("User", "Let's create a story about a space
adventure."))


for turn in range(6):
    next_agent = manager.select_next_speaker(conversation)
role_name = next_agent.role
    reply = next_agent.respond(conversation)
    conversation.append((role_name, reply))
    print(f"{role_name}: {reply}\n")
```

In this example, ConversationManager.select_next_speaker could simply rotate through the list, or use more complex logic. Each agent's respond function would likely use an LLM prompt with the conversation transcript as context. All agents and the manager see the same conversation list, ensuring they have a common worldview of what has been said. This pattern creates a collaborative dialogue, useful for brainstorming, role-playing, or any scenario where *multiple experts need to discuss*. The group

chat ends either after a fixed number of turns or when a termination condition is met (perhaps decided by the manager agent).

References: The definition of the group chat pattern and its turn-taking mechanism is described in AutoGen's documentation . Realistic enterprise use of this pattern is exemplified by the Microsoft Semantic Kernel sample, where specialized Copilot agents chat together to handle complex user requests . Research into multi-agent communications like CAMEL provides insight into how role-based chat between AI agents can solve tasks autonomously.

Handoffs

Description: *Handoffs* is a design pattern in which one agent can delegate or transfer a task/interaction to another agent during run-time. In a conversation context, this often means the current agent stops handling the user's query and "hands off" control to a different agent better suited for the next part of the task. This pattern enables dynamic role switching and specialization within a single session. The idea was notably introduced by OpenAI's experimental framework Swarm, which treats handoffs as a first-class mechanism for lightweight agent orchestration . The purpose is to allow a seamless transition between agents – much like a customer service chatbot handing a customer over to a human agent, except here it could be AI-to-AI handoff (or AI to human, or vice versa) depending on expertise needed.

Implementation Strategies & Tools: One common implementation for handoffs is through structured function calls or messages that specify the next agent. In Swarm, an agent can decide at any point to return a new agent (or its identifier) as the result of a function call, effectively telling the system "switch to this agent now" . From the outside, it looks like the conversation continues, but under the hood the system has swapped the persona/logic handling it. Concretely, Swarm's abstraction treats each agent as having an Agent class with certain instructions/tools, and if it wants to delegate, it calls a special *handoff function* that instantiates or selects another agent to take over . Other frameworks might implement handoff via a controller agent: e.g. a *Triage Agent* reads the user request, then forwards it to the appropriate specialist agent by sending that agent a message with the full context. In AutoGen's event-driven approach, a handoff can be done by publishing the conversation (as a UserTask message) to a topic that a target agent is listening on, thereby effectively passing control . Tools like **LangChain** or custom orchestrators can also be coded to achieve this: e.g., you could write a logic like if query category == 'refund': next_agent = RefundAgent; else if 'sales': next_agent = SalesAgent. The key technical challenge is maintaining the **conversation state** across the switch – ensuring the new agent knows the history. Swarm addresses this by keeping a single shared message history that the next agent can see, and possibly altering the system prompt to reflect the new agent's identity .

All this can happen within the same chat window, with the user possibly unaware that different "agents" are responding at different times – ideally the transition is smooth. This is analogous to how real call centers escalate calls, but here it's AI agents trading off. OpenAI's Swarm framework (open-sourced on GitHub) provides educational examples of this pattern, such as a triage_agent that demonstrates routing to the right agent using handoffs . Another example from the Swarm README describes a scenario like a game where a user can say "let me speak to the wise sage," and the system will hand off the conversation from a generic guide agent to a specialized "sage" agent for the next part . In enterprise chatbot solutions,

bot-to-human handoff is a well-known practice (when the bot encounters something it can't handle). The same concept extends to bot-to-bot handoff for chaining skills. For instance, a virtual assistant might initially handle a request but then hand off to a scheduling bot for calendar specifics, then back – though this could also be seen as a form of tool usage or subroutine, it aligns with the handoff architecture if implemented as distinct agents.

Code Example: Here is pseudocode that sketches how a handoff might be orchestrated in a conversation loop. We have an initial agent (current_agent), which can return a special indication that the next agent should take over (we'll use a tuple (response, next_agent) for illustration):

```
current_agent = TriageAgent()
conversation_history = []

while True:
    user_input = get_user_input()
    conversation_history.append(("User", user_input))
    reply, next_agent = current_agent.handle(user_input,
conversation_history)
    conversation_history.append((current_agent.name, reply))
    print(f"{current_agent.name}: {reply}")
    if next_agent is not None:
        # Switch to the new agent for subsequent interactions
        print(f"**Handing off to {next_agent.name}**")
        current_agent = next_agent
    # break condition (if conversation ended by an agent or user)
    if current_agent.finished or user_input in ("bye", "exit"):
        break
```

In this snippet, TriageAgent.handle() might analyze the input and decide next_agent = RefundAgent() (for example) along with a reply like "Sure, I can help with that refund." The loop then switches current_agent. The conversation history is preserved and passed along, so the RefundAgent can see what the user asked initially. The handoff is signaled here by printing a notice, but in a real system the user might just see a continuous conversation. This logic could be implemented with OpenAI function calling: e.g., the triage agent's response could include a function call payload "action": "handoff_to_refund_agent", which the system intercepts to instantiate the RefundAgent.

References: OpenAI's Swarm project introduced and exemplified this pattern: "an Agent can at any point choose to hand off a conversation to another Agent" . The AutoGen documentation on Handoffs further explains how an agent can delegate a task to others via a special tool call, and provides a customer support use-case as described above . These sources show the practical benefits of modularizing agent responsibilities and dynamically routing conversations.

Mixture of Agents

Description: *Mixture of Agents* is a pattern inspired by the *mixture-of-experts* paradigm in neural networks, wherein multiple agents are organized in layers and an orchestrator funnels a task through these layers to progressively refine the outcome . There are typically two types of components in this pattern: a single orchestrator (aggregator) agent and multiple worker agents divided into several layers. All workers in a layer work in parallel on the input (which may include the aggregated outputs from the previous layer), and then the orchestrator prepares the collective output as input to the next layer. In essence, it's a *feed-forward multi-agent architecture*: the query passes through layer 1 agents, their outputs are combined, then layer 2 agents use that to produce new outputs, and so on . By the final layer, the orchestrator aggregates all results into one answer. The purpose of MoA is to harness diverse expertise or model strengths – e.g., multiple small LLMs working together can outperform a single large LLM on certain tasks, as each agent contributes partial insights that get merged.

Implementation Strategies & Tools: The orchestrator agent is the key to implementing MoA. It must dispatch the task to all agents in layer 1, wait for their results, possibly synthesize those results into a new prompt, send that to all agents in layer 2, and so forth . A straightforward strategy is to use synchronous rounds: in each round corresponding to a layer, the orchestrator triggers all agents (e.g., via asynchronous calls or parallel messaging) and collects their responses. The combination step can be as simple as concatenating all responses (perhaps prefaced by a system instruction about synthesizing them) or as complex as feeding them into another model to summarize. AutoGen's implementation uses the send_message() API to directly send messages to worker agents and gather replies, making it easier to manage cancellations or errors in each round . They also show how to use dataclasses for message types like WorkerTask and WorkerTaskResult to structure communication . Tools that support *barrier synchronization* (waiting for multiple agents) are handy – for instance, using Python asyncio.gather to await multiple agent calls concurrently in each layer. In a distributed setting, one could use a topic per layer: orchestrator publishes to "layer1" topic, all layer1 agents subscribe and respond, orchestrator collects and publishes to "layer2" topic, etc. The pattern requires that all agents in a given layer can work independently (no inter-communication at same layer, usually) and that their outputs are combinable. It's common to have homogeneous agents (all similar capability) in each layer, but one could also have heterogeneous ones if each provides a different facet of an answer. The final aggregation could be a majority vote, a weighted vote, or an LLM aggregating content.

Code Example: The following pseudocode demonstrates a simple two-layer Mixture of Agents. In layer 1, we have two worker agents that each produce a draft answer to a question. In layer 2, we have one agent that takes all drafts and produces a final answer. The orchestrator manages the flow between layers:

```
# Define two layers of agents
layer1_agents = [Agent("Agent1_L1"), Agent("Agent2_L1")]
layer2_agents = [Agent("Agent1_L2")]  # could be multiple in layer2 as
well

question = "Q: What are the benefits of solar energy?"
```

```
# Layer 1: dispatch question to all layer1 agents
drafts = []
for agent in layer1_agents:
    draft = agent.generate_answer(question)
    drafts.append(draft)
    print(f"{agent.name} draft: {draft}")
# Orchestrator combines drafts (simple concatenation in this example)
combined_input = "Other agents answered:\n" + "\n---\n".join(drafts)
# Layer 2: dispatch combined input to layer2 agents
final_outputs = []
for agent in layer2_agents:
    final = agent.generate_answer(combined_input)
    final_outputs.append(final)
    print(f"{agent.name} final output: {final}")
# Aggregation: if multiple in layer2, could vote or average. Here just
take the single final.
final_answer = final_outputs[0]
print("Final Answer:", final_answer)
```

In this toy example, each Layer1 agent independently answers the question. The orchestrator then creates a prompt for Layer2 agents that includes all Layer1 answers (perhaps with instructions to synthesize or pick the best). The Layer2 agent(s) then produce improved answers. If there were multiple Layer2 agents, an aggregator (which could be part of orchestrator or another agent) might then decide on the final answer (e.g., via majority vote or by asking an LLM to pick the most coherent answer). The power of this pattern is that even relatively weak agents, by collaborating in layers, can produce a strong final result through **iterative refinement** .

References: Microsoft's AutoGen documentation provides a clear description of the Mixture-of-Agents pattern and its implementation , referencing the original work by Wang et al. The arXiv paper *"Mixture-of-Agents Enhances Large Language Model Capabilities"* offers the research foundation and reports empirical gains , with code available for reference. These resources highlight how layering and orchestration in MoA can lead to emergent capabilities beyond a single model.

Multi-Agent Debate

Description: *Multi-Agent Debate* is a pattern where multiple agents engage in an iterative **discussion or debate**, often to improve reasoning accuracy or to evaluate answers. In a debate setting, agents exchange their points of view over several turns, critiquing or building on each other's responses, and by the end, the system decides on a final answer (for example, via voting or a judge agent). This is analogous to a panel of experts deliberating on a question. Each agent may have the same role (symmetric debate) or different roles (e.g., proposer vs. critic). The key idea is that through confrontation of ideas and error

correction across agents, the final result will be more robust. Each round, agents refine their responses based on what others said in the previous round .

Implementation Strategies and Tools: Implementing a debate requires a mechanism for turn-based multi-turn interactions and a way to share information among agents. Often, a central *moderator/aggregator* agent is used to coordinate the rounds: it poses the initial question to all solver agents, collects their answers, then for each subsequent round, it may broadcast each agent's last answer to all the other agents as additional context . In AutoGen's approach, they utilize the publish-subscribe system with specific topics to simulate a communication topology. Another approach is simply a loop in code: have each agent take turns reading the transcript and appending a new message (like a group chat, but with a structured objective). After a fixed number of debate rounds, an aggregation step happens. This could be majority voting among the agents' final answers , or using a designated *judge agent* that evaluates which answer is best. In OpenAI's early formulation of AI debate, they actually involved a human judge to pick the winning argument , but in automated systems, one of the agents or a separate model can serve as the judge. Tools that help include any multi-turn dialogue framework (since a debate is basically a multi-turn chat with a particular format). The debate can be fully connected or have a *sparse communication topology* to reduce noise . Recent research suggests that limiting each agent to see only some others can achieve similar benefits at lower cost . Therefore, how messages are routed is an important design choice in multi-agent debate.

Code Example: Pseudocode for a simple multi-agent debate with three agents might look like this. Each agent will attempt to solve the problem, then we iterate a couple of rounds where they exchange their latest answers:

```
agents = [Agent("Agent1"), Agent("Agent2"), Agent("Agent3")]
question = "Question: What is 79 * 46?"  # a reasoning task
# Initial answers from each agent
answers = {}
for agent in agents:
    answers[agent.name] = agent.solve(question)
print("Initial answers:", answers)

# Debate rounds - each round agents update their answer after seeing
others' answers
for round in range(1, 3):  # 2 rounds of debate
    for agent in agents:
        # Agent sees answers from others (could be all or a subset)
        other_answers = {name: ans for name, ans in answers.items() if
name != agent.name}
        answers[agent.name] = agent.refine_answer(question,
other_answers)
    print(f"After round {round}: {answers}")
```

```
# Aggregation (e.g., majority vote or pick first if consensus)
final_answers = list(answers.values())
final_answer = max(set(final_answers), key=final_answers.count)  # majority vote
print("Final answer by majority vote:", final_answer)
```

Each of these architectural patterns – from *Concurrent Agents* to *Multi-Agent Debate* – offers a different trade-off in structure and complexity. In practice, complex AI systems may combine multiple patterns. For example, a large system might use a Concurrent + Handoff approach: start multiple agents concurrently to handle different parts of a user request, then have a manager agent hand off the conversation to whichever agent's results look most promising. Or a system might run a Debate among a few agents and then use a Reflection pattern (another pattern, not detailed above, where an agent analyzes the debate outcome) to finalize an answer. These patterns are not mutually exclusive and often a robust multi-agent architecture will draw on several. As research and industry applications of multi-agent systems continue to grow (spanning companies and labs globally), the design patterns discussed here serve as proven templates. They help engineers and researchers structure agent interactions in a way that is scalable, maintainable, and aligned with the problem at hand – whether it's orchestrating enterprise workflows with AI assistants or pushing the state of the art in collaborative intelligence.

Comparative Analysis

Pattern	Coordination Style	Scalability
Concurrent	Parallel processing	High
Sequential	Linear pipeline	Moderate
Group Chat	Turn-based discussion	Variable
Handoffs	Delegation hierarchy	High
Mixture of Agents	Layered refinement	High

Addressing Autograder Limitations with Multi-Agent Frameworks

The principles and successes of MAS in software engineering can be leveraged to tackle the challenges faced by traditional auto-graders:

- **Enhanced and Formative Feedback**: By employing multiple agents with specialized roles (e.g., a syntax checker [37], a style checker [37], a test coverage checker [37], and a metrics checker [39]), a multi-agent autograding system can provide more comprehensive feedback, addressing not only functional correctness but also code quality and adherence to best practices [40]. The integration of LLMs can enable the generation of conceptual-level feedback from low-level program patches [13]. Systems like TILE [1] and the platform used with Quarterfall [40] aim to provide "help for self-help" through customized feedback messages.
- **Improved Bug Detection and Diagnosis**: Drawing inspiration from fault localization systems like AGENTFL [16], a multi-agent autograder could employ agents to analyze student code from different perspectives (e.g., a testing agent, a logic analysis agent) to identify a wider range of errors and provide more insightful diagnoses.
- **Increased Customization and Flexibility**: The modular nature of multi-agent frameworks allows for greater flexibility in designing autograding systems that can be tailored to specific instructor needs and course policies [42]. Different agents with specific functionalities can be integrated or configured to support diverse grading strategies [11].
- **Advanced Testing and Edge Case Identification**: Similar to the work in automated test generation [24], a multi-agent system could include agents dedicated to generating and refining unit tests to achieve higher code coverage and identify edge cases that traditional test suites might miss [25].
- **Integration and Extensibility**: Frameworks like AutoGen [25] provide a basis for building multi-agent systems with built-in code execution and iterative refinement capabilities, facilitating the development of extensible and integrable autograding solutions [40].

Considerations and Future Directions

While the application of multi-agent frameworks to autograding holds significant promise, several considerations and areas for future research exist. Dynamically adapting feedback based on student attempts [44], ensuring robustness [44] and addressing potential biases within LLM powered agents [45] are crucial aspects to consider. The effective design of agent interactions [46], the balance between agent autonomy and human oversight [48], and the development of appropriate evaluation metrics for collaborative assessment processes [50] also warrant further investigation. Despite the increasing interest in using LLMs in education [51], the current lack of widespread instructor experience with these tools in auto-grading highlights a need for more research and practical implementations in this area [9].

Conclusion

The challenges associated with traditional auto-graders, particularly in providing nuanced feedback and adapting to diverse needs, can potentially be addressed by leveraging the power of multi-agent frameworks powered by large language models. By drawing on the advancements in MAS for various software engineering tasks, it is possible to design more intelligent, customizable, and effective auto-grading systems that can enhance the learning experience for students and reduce the workload for instructors in computing education. Future research should focus on designing, implementing, and evaluating such multi-agent autograding systems to fully realize their potential.

References

[1] Joachim Breitner, Martin Hecker, and Gregor Snelting. "Der Grader Praktomat". In: Automatisierte Bewertung in der Programmierausbildung. Ed. by Oliver J. Bott et al. Digitale Medien in der Hochschullehre 6. Waxmann Verlag GmbH, 2017, pp. 159–172.

[2] Anderson Pinheiro Cavalcanti et al. "Automatic feed-back in online learning environments: A systematic literature review". In: Computers and Education: Artificial Intelligence 2 (2021), p. 100027.

[4] Michael Goedicke, Michael Striewe, and Moritz Balz. Computer aided assessments and programming exercises with JACK. Tech. rep. ICB-Research Report, 2008.

[6] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. "A Systematic Literature Review of Automated Feedback Generation for Programming Exercises". In: ACM Trans. Comput. Educ. 19.1 (2018).

[7] N. Singer, "The Hard Part of Computer Science? Getting Into Class," January 2019.

[9] S. Mirhosseini, A. Z. Henley, and C. Parnin, "What is your biggest pain point? an investigation of cs instructor obstacles, workarounds, and desires," in Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, 2023.

[11] B. Anderson, M. Henz, and K.-L. Low, "Community-driven course and tool development for cs1," in Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, 2023.

[13] D. Cambaz and X. Zhang, "Use of ai-driven code generation models in teaching and learning programming: a systematic literature review," in Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1, 2024.

[14] A. Hellas, J. Leinonen, S. Sarsa, C. Koutcheme, L. Kujanpää, and J. Sorva, "Exploring the responses of large language models to beginner programmers' help requests," in Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1, 2023.

[15] Z. Fan, S. H. Tan, and A. Roychoudhury, "Concept-based automated grading of cs-1 programming assignments," in Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023.

[16] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," IEEE Transactions on Software Engineering, vol. 42, no. 8, pp. 707–740, 2016.

[20] M. Schreier, Qualitative content analysis in practice. Sage publications, 2012.

[21] GrayC

[22] Csmith

[24] YARPGen

[25] TypeFuzz

[35] Securify

[36] Vandal

[37] Zeus

[39] V. Wüstholz and M. Christakis. Harvey: A greybox fuzzer for smart contracts. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020.

[40] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong, M. Zhang, J. Wang, S. Jin, E. Zhou, et al. "The rise and potential of large language model based agents: A survey." arXiv preprint arXiv:2309.07864, 2023.

[42] Michael Goedicke, Michael Striewe, and Moritz Balz. Computer aided assessments and programming exercises with JACK. Tech. rep. ICB-Research Report, 2008.

[44] C. S. Xia, Y. Wei, and L. Zhang. "Automated program repair in the era of large pre-trained language models." In Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery, 2023.

[45] J. C. Heller. Catch-22: a novel, volume 4. Simon and Schuster, 1999.

[46] Y. Zhang, J. Yang, Y. Yuan, and A. C.-C. Yao. "Cumulative reasoning with large language models." arXiv preprint arXiv:2308.04371, 2023.

[48] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. "Large language model based multi-agents: A survey of progress and challenges." arXiv preprint arXiv:2402.01680, 2024.

[50] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, et al. "A survey of large language models." arXiv preprint arXiv:2303.18223, 2023.

[51] Gustavo Pinto et al. "Large Language Models for Education: Grading Open-Ended Questions Using ChatGPT." arXiv:2307.16696, 2023.