Mirador: An Orchestrated AI Chaining Framework

Authors: Matthew Scott and Claude Anthropic
Date: May 11, 2025

Abstract

This paper presents Mirador, a novel orchestration framework for chaining specialized AI models into cohesive workflows. Named after architectural viewpoints that provide panoramic perspectives, Mirador enables multiple specialized AI models to collaborate on complex tasks, with each model contributing its unique expertise to a larger process. Unlike monolithic approaches that rely on a single large language model, Mirador adopts a modular architecture where purpose-built models handle specific aspects of a task, creating results that exceed what any single model could produce. This paper details Mirador's architecture, implementation, and capabilities, showcasing how it addresses fundamental challenges in AI orchestration through a balance of automation and human oversight. By providing transparent process visibility, session persistence, and extensible domain adaptations, Mirador represents a significant advancement in practical AI system design. The philosophical underpinnings of Mirador suggest a future direction for AI systems that favors specialization and collaboration over generalization, mirroring successful patterns found in human creative and technical endeavors.

1. Introduction

The development of large language models has led to remarkable capabilities across numerous domains. However, these models often represent a compromise between breadth and depth - while they can perform many tasks adequately, they rarely excel at specialized functions compared to purpose-built systems. Mirador addresses this limitation by enabling multiple specialized AI models to work together on complex tasks, with each model contributing its expertise within a structured workflow.

Mirador stems from the recognition that the most sophisticated human endeavors - from software development to creative production - often involve multiple specialists working in sequence, with each adding their unique perspective and skills. The framework implements this principle by orchestrating "personas" - AI models with specific roles and capabilities - in configurable chains, with human checkpoints to ensure quality control.

This paper explores the architecture, implementation, and capabilities of Mirador, focusing on its practical applications in software development, creative work, and specialized domains such as guitar technique analysis. We demonstrate how Mirador's modular approach enables more sophisticated AI applications compared to single-model approaches, while maintaining human oversight at critical junctures.

2. Technical Foundation

2.1 Architecture Overview

Mirador's architecture consists of four primary components:

1. **Core Framework:** The central orchestration system that manages the flow between AI models
2. **Persona System:** Configurable AI roles with specific functions (e.g., master_coder, code_reviewer)
3. **Session Management:** Persistent storage of execution history with resumability
4. **API Layer:** RESTful interface for remote access and integration

At its core, Mirador is built around the concept of a processing chain - a sequence of specialized AI models (personas) that process content iteratively, with each model building upon the output of the previous one. Figure 1 illustrates this architecture.

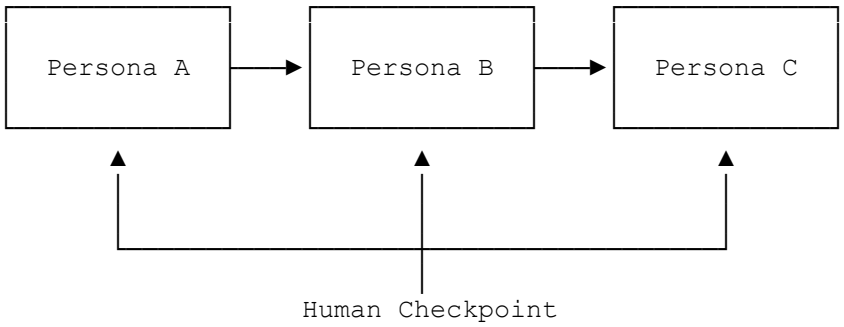

Figure 1: Mirador processing chain with human checkpoints

2.2 Persona System

The persona system is the foundation of Mirador's specialization approach. Each persona represents a specialized AI model configured for a specific role in the processing chain. Personas are defined with:

- A unique identifier (e.g., master_coder)
- An associated Ollama model (e.g., master-coder:latest)
- A role description (e.g., "Generate baseline implementation")
- A prompt template that structures how input is presented to the model

This configuration-driven approach allows for the flexible definition of roles without requiring code changes. New personas can be added by defining their parameters in the configuration file.

2.3 Session Management

Mirador implements robust session management to maintain context across runs and enable process resumption. Each execution session is assigned a unique identifier (typically a timestamp) and maintains:

- Inputs and outputs for each persona in the chain
- Execution timing information
- Process metadata
- A summary of the entire execution

This persistent session storage enables users to:

- Resume interrupted processing
- Review previous outputs
- Continue processing with modified inputs
- Track progress across multiple iterations

2.4 API Layer

The API layer exposes Mirador's functionality through a RESTful interface, enabling:

- Remote execution of chains
- Individual persona invocation
- Session management and retrieval
- Configuration access

This layer transforms Mirador from a command-line utility to a service that can be integrated with other applications, extending its reach beyond direct terminal interaction.

3. Implementation

3.1 Core Framework

The heart of Mirador is its framework.py module, which implements the AIFramework class responsible for orchestrating the entire process. The implementation focuses on:

- Chain execution management
- Persona invocation with appropriate context
- Session persistence
- Error handling and recovery
- Human interaction points

The framework uses a streaming approach to model interaction, providing real-time feedback during execution rather than waiting for complete responses. This enables more responsive user experiences, especially with longer-running processes.

```
def run_chain(self, input_text, start_at=None, end_at=None, interactive=True,
```

```
              persona_parameters=None, skip_optional=False,
continue_on_error=False):
    """
    Run the complete chain or a segment of it.

    Args:
        input_text (str): The input text to process
        start_at (str, optional): Persona ID to start at
        end_at (str, optional): Persona ID to end at
        interactive (bool): Whether to run in interactive mode with user
prompts
        persona_parameters (dict, optional): Dict mapping persona IDs to
parameter dicts
        skip_optional (bool): Whether to automatically skip optional nodes
        continue_on_error (bool): Whether to continue chain execution on
errors

    Returns:
        dict: Dictionary of outputs from each persona
    """
    # Implementation details...
```

3.2 Model Specialization

Mirador leverages Ollama's model customization capabilities through modelfiles, which define specialized versions of base models like Llama3 and DeepSeek. These specializations include:

```
FROM llama3
PARAMETER num_ctx 8192
PARAMETER num_gpu 99
PARAMETER num_thread 12
PARAMETER temperature 0.7
SYSTEM """You are a world-class Python developer with expertise in efficient
algorithms and data structures. Your task is to generate clean, working code
implementations based on requirements. Focus on:
1. Correctness - code should work as expected
2. Efficiency - optimize for speed and resource usage
3. Readability - clear variable names and structure
4. Error handling - proper validation and exception management
5. Standards compliance - follow PEP 8 guidelines

Your code should be complete and runnable. Include docstrings and comments
where appropriate. Do not use placeholder code - implement full solutions."""
```

Example modelfile for master-coder persona

Each persona's modelfile is carefully crafted to enhance specific capabilities, using:

- Appropriate context window sizes
- Optimized parameters for the specific task
- Detailed system prompts that focus the model on its specialized role
- Task-specific examples and guidelines

3.3 Session Persistence

Mirador implements session persistence through a hierarchical file structure:

```
ai_framework/
├── outputs/
│   ├── 20250511-133327/
│   │   ├── initial_input.txt
│   │   ├── master_coder_input.txt
│   │   ├── master_coder_output.txt
│   │   ├── code_reviewer_input.txt
│   │   ├── code_reviewer_output.txt
│   │   └── summary.md
```

This structure enables:

- Complete traceability of the execution process
- Simple inspection of intermediate outputs
- Recovery from interruptions
- Analysis of performance and results

3.4 API Implementation

The API is implemented using Flask, providing endpoints for:

- Chain execution (`/api/run`)
- Individual persona execution (`/api/run/{persona_id}`)
- Session listing (`/api/sessions`)
- Session details (`/api/sessions/{session_id}`)
- Output retrieval (`/api/sessions/{session_id}/outputs/{persona_id}`)
- Configuration access (`/api/personas`)

The API layer maintains consistency with the command-line interface while enabling remote and programmatic access.

4. Mirador in Action: Case Studies

4.1 Code Generation and Improvement

The primary case study demonstrating Mirador's capabilities involves generating, enhancing, and documenting a Python function to calculate Fibonacci numbers. This case study showcases the full chain execution with all specialized personas.

Initial Prompt

```
Create a Python function to calculate the Fibonacci sequence
```

Master Coder Output

The master-coder persona generated a baseline implementation with proper documentation and error handling:

```
def fibonacci(n):
    """
    Calculate the nth number in the Fibonacci sequence.

    Args:
        n (int): The position of the number in the sequence.

    Returns:
        int: The nth number in the Fibonacci sequence.
    """
    if not isinstance(n, int) or n < 0:
        raise ValueError("n must be a non-negative integer")

    if n <= 1:
        return n

    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b

    return b
```

Code Reviewer Enhancement

The code-reviewer persona identified potential issues with large values and enhanced the implementation:

```
def fibonacci(n):
    """
    Calculate the nth number in the Fibonacci sequence.

    Args:
        n (int): The position of the number in the sequence.

    Returns:
        int: The nth number in the Fibonacci sequence.
    """
    if not isinstance(n, int) or n < 0:
        raise ValueError("n must be a non-negative integer")

    if n <= 1:
        return n

    a, b = 0, 1
    for _ in range(2, min(n + 1, 92)):
        a, b = b, a + b

    return b
```

Note the addition of `min(n + 1, 92)` to handle Python's integer limits.

File Reviewer Documentation

The file-reviewer persona provided comprehensive documentation and usage explanation:

Summary: This code calculates and returns the nth number in the Fibonacci sequence. The Fibonacci sequence is a series of numbers where each subsequent number is the sum of the previous two, starting from 0 and 1.

Purpose and Utility: The purpose of this code is to provide an efficient way to calculate Fibonacci numbers up to a certain position (n). This can be useful in various applications such as mathematics, finance, or computer science.

Creative Applications

The creative-entrepreneur persona identified multiple real-world applications:

1. **Financial Trading Tool:** Calculating Fibonacci retracement levels for technical analysis
2. **Procedural Content Generation:** Creating natural patterns in gaming environments
3. **Educational Math Tools:** Visualizing mathematical sequences and patterns
4. **Music Composition:** Generating compositions based on mathematical patterns
5. **Botanical Growth Modeling:** Simulating natural growth patterns in plants

User Interface Design

The ux-designer persona proposed both CLI and GUI implementations:

```
# GUI Implementation
import tkinter as tk
from tkinter import ttk, messagebox

def fibonacci(n):
    # Implementation...

def main():
    root = tk.Tk()
    root.title("Fibonacci Calculator")

    # UI implementation details...

    root.mainloop()

if __name__ == "__main__":
    main()
```

This case study demonstrates how Mirador transforms a simple request into a comprehensive solution with implementation, optimization, documentation, application ideas, and user interface design - each aspect handled by a specialized model.

4.2 Guitar Technique Analysis

Another notable case study involves Mirador's domain-specific application for guitar technique analysis, showcasing the framework's adaptability to specialized knowledge domains.

The guitar-expert persona demonstrated detailed knowledge of guitar techniques, providing:

Economy picking is a technique that enhances guitar playing efficiency by combining downstrokes, upstrokes, and finger techniques to minimize pick hand movement. Here's an organized overview:

What is Economy Picking?

- **Definition**: A method where guitarists use the most efficient strokes for each note, often using consecutive downstrokes or upstrokes when beneficial.
- **Differentiation from Sweep Picking**: Unlike sweep picking, which involves sweeping across multiple strings in one stroke, economy picking focuses on efficiency within individual phrases.

Applications in Technical Death Metal

- **Usage**: Commonly used in fast riffs and solos where quick transitions between notes are essential. It allows for smoother execution of complex patterns without the constraints of alternate picking.
- **Integration with Other Techniques**: Can be combined with tapping, hammer-ons, or pull-offs to create intricate and fluid passages.

This example demonstrates how Mirador can be extended to highly specialized domains beyond general software development, leveraging domain-specific models to provide expert-level guidance.

5. Command-Line Interface Examples

Mirador provides a flexible command-line interface that supports various usage patterns. Table 1 presents a comprehensive overview of command examples and their applications.

Table 1: Mirador Command-Line Interface Examples

Command	Description
`python3 -m ai_framework.framework --prompt "Create a Python function"`	Basic chain execution with a simple prompt
`python3 -m ai_framework.framework --file input.txt`	Process content from a file
`python3 -m ai_framework.framework --prompt "..." --start code_reviewer`	Start chain at a specific persona

Command	Description
`python3 -m ai_framework.framework --prompt "..." --end master_coder`	End chain at a specific persona
`python3 -m ai_framework.framework --prompt "..." --auto`	Run in non-interactive mode
`python3 -m ai_framework.framework --prompt "..." --skip-optional`	Skip optional personas
`python3 -m ai_framework.framework --list`	List all available personas
`python3 -m ai_framework.framework --continue 20250511-133327`	Continue a previous session

For more advanced use cases:

```
# Run a chain with specific persona parameters
cat <<EOF > params.json
{
  "master_coder": {
    "temperature": 0.5,
    "top_p": 0.9
  },
  "code_reviewer": {
    "temperature": 0.3
  }
}
EOF

python3 -m ai_framework.framework --prompt "Implement a binary search tree" --parameters params.json

# Process a large document with domain-specific analysis
python3 -m ai_framework.framework --file /path/to/research_paper.txt --start rag_node --end creative_entrepreneur
```

These examples demonstrate the flexibility and power of Mirador's command-line interface, enabling both simple and complex workflows.

6. Development Journey

The development of Mirador represents a collaborative journey between human creativity and AI assistance. This section explores the key moments and interactions that transformed an initial concept into a robust framework.

6.1 Genesis

Mirador began as Matthew Scott's vision for a system that could chain specialized AI models together. The initial implementation was focused on the core functionality - executing a sequence of models with human checkpoints - but lacked some of the more sophisticated features like comprehensive session management and API access.

The project was born out of a recognition that while large language models excel at many tasks, truly exceptional results often require specialized expertise. By combining purpose-built models in a structured workflow, Mirador aimed to achieve results beyond what any single model could produce.

6.2 Pivotal Interactions

Several key moments defined the collaborative development process:

1. **Framework Enhancement:** The foundational enhancement to the AIFramework class added robust session management, enabling users to resume work and maintain context across sessions.
2. **API Development:** The creation of a RESTful API interface transformed Mirador from a command-line tool to a service that could be integrated with other applications.
3. **Guitar Studio Implementation:** The development of a domain-specific extension demonstrated Mirador's adaptability to specialized knowledge domains beyond software development.
4. **Comprehensive Testing:** The successful execution of the complete chain for the Fibonacci function case study validated the framework's end-to-end capabilities.

Throughout this process, Matthew exhibited initial hesitation about entrusting his fragile project to external assistance. He had already developed a working prototype and was concerned about maintaining its integrity while expanding its capabilities. Research using Perplexity helped establish confidence that Claude Code and Ollama could work together effectively, leading to a productive collaboration.

The excitement of seeing Mirador transform from a collection of one-off scripts into a cohesive system was palpable. The realization that the framework could not only generate code but also optimize it, document it, identify applications, and design interfaces - all through specialized models working in concert - represented a significant breakthrough.

7. Future Enhancements

While Mirador already provides comprehensive capabilities, several potential enhancements could further extend its functionality and usability.

7.1 Core Framework Enhancements

1. **Parallel Processing:** Enable concurrent execution of independent personas to reduce overall processing time
2. **Branching Workflows:** Support conditional execution paths based on intermediate results
3. **Feedback Loops:** Allow iterative refinement through cyclic processing patterns
4. **Performance Optimization:** Implement caching and result reuse for common operations
5. **Enhanced Error Recovery:** Provide more sophisticated mechanisms for handling and recovering from failures

6. **Customizable Checkpoints:** Allow finer-grained control over when human intervention occurs

7.2 UI and Visualization

1. **Web Interface:** Develop a comprehensive web application for managing Mirador workflows
2. **Chain Visualization:** Create interactive visualizations of processing chains and their execution
3. **Result Comparison:** Enable side-by-side comparison of outputs from different sessions or configurations
4. **Real-time Monitoring:** Provide dashboards for monitoring active processing
5. **Interactive Editor:** Create a specialized editor for updating intermediate outputs

7.3 Integration Capabilities

1. **Webhooks:** Implement webhook notifications for process completion
2. **OAuth Authentication:** Add secure authentication for API access
3. **SDK Development:** Create language-specific SDKs for common programming languages
4. **Plugin System:** Develop a formal plugin architecture for extending functionality
5. **Third-party Tool Integration:** Enable integration with common development tools (GitHub, JIRA, etc.)
6. **Docker Containerization:** Package Mirador for easy deployment in containerized environments

7.4 Knowledge Management

1. **Persistent Knowledge Base:** Implement a vector database for maintaining domain knowledge
2. **Cross-session Learning:** Enable knowledge transfer between separate processing sessions
3. **Collaborative Editing:** Support multiple users working on the same session
4. **Version Control:** Implement version tracking for outputs and chain configurations
5. **Annotations:** Allow users to annotate outputs with explanations and clarifications

7.5 Domain-Specific Extensions

1. **Scientific Research:** Extend Mirador for scientific literature analysis and experiment design
2. **Legal Document Processing:** Create specialized personas for legal document drafting and analysis
3. **Financial Analysis:** Develop personas for financial modeling and investment research
4. **Creative Writing:** Implement specialized workflows for fiction, poetry, and screenplay development

5. **Educational Content:** Create personas focused on developing learning materials and assessments

These enhancements represent a roadmap for Mirador's future development, though it's important to note that they should be implemented incrementally with thorough testing and validation. As emphasized, Mirador is not intended for commercial distribution but rather as a personal exploration of AI orchestration principles.

8. Conclusion

Mirador demonstrates a novel approach to AI orchestration that combines specialized models in structured workflows. By embracing specialization over generalization, it achieves results that exceed what any single model could produce, while maintaining human oversight at critical junctures.

The framework's architecture - with its modular personas, persistent sessions, and flexible API - provides a solid foundation for sophisticated AI applications across diverse domains. From software development to creative work to specialized knowledge areas, Mirador offers a powerful tool for leveraging AI in complex workflows.

The development journey itself represents a testament to the potential of human-AI collaboration. What began as a fragile concept evolved into a robust framework through iterative refinement and thoughtful implementation. The result is not just a collection of scripts but a cohesive system that embodies a philosophical perspective on AI orchestration - one that values specialized expertise, transparent processes, and human guidance.

Mirador ultimately serves as both a practical tool and a conceptual model for future AI systems. By demonstrating the power of specialization, orchestration, and collaboration, it points toward a future where AI systems work in concert - with each other and with humans - to tackle complex challenges with unprecedented sophistication and nuance.

References

1. Scott, M. (2025). Mirador AI Framework: Core implementation. GitHub Repository.
2. Anthropic. (2024). Claude: A next-generation AI assistant. Anthropic.
3. Ollama. (2024). Ollama: Run large language models locally. GitHub Repository.
4. Lewis, M., et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv preprint arXiv:2005.11401.
5. Wei, J., et al. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv preprint arXiv:2201.11903.

Acknowledgements: The authors would like to acknowledge the contributions of the open-source AI community, particularly the developers of Ollama and the foundational models that

enable Mirador's specialized personas. Special thanks to the early testers who provided valuable feedback on Mirador's functionality and usability.