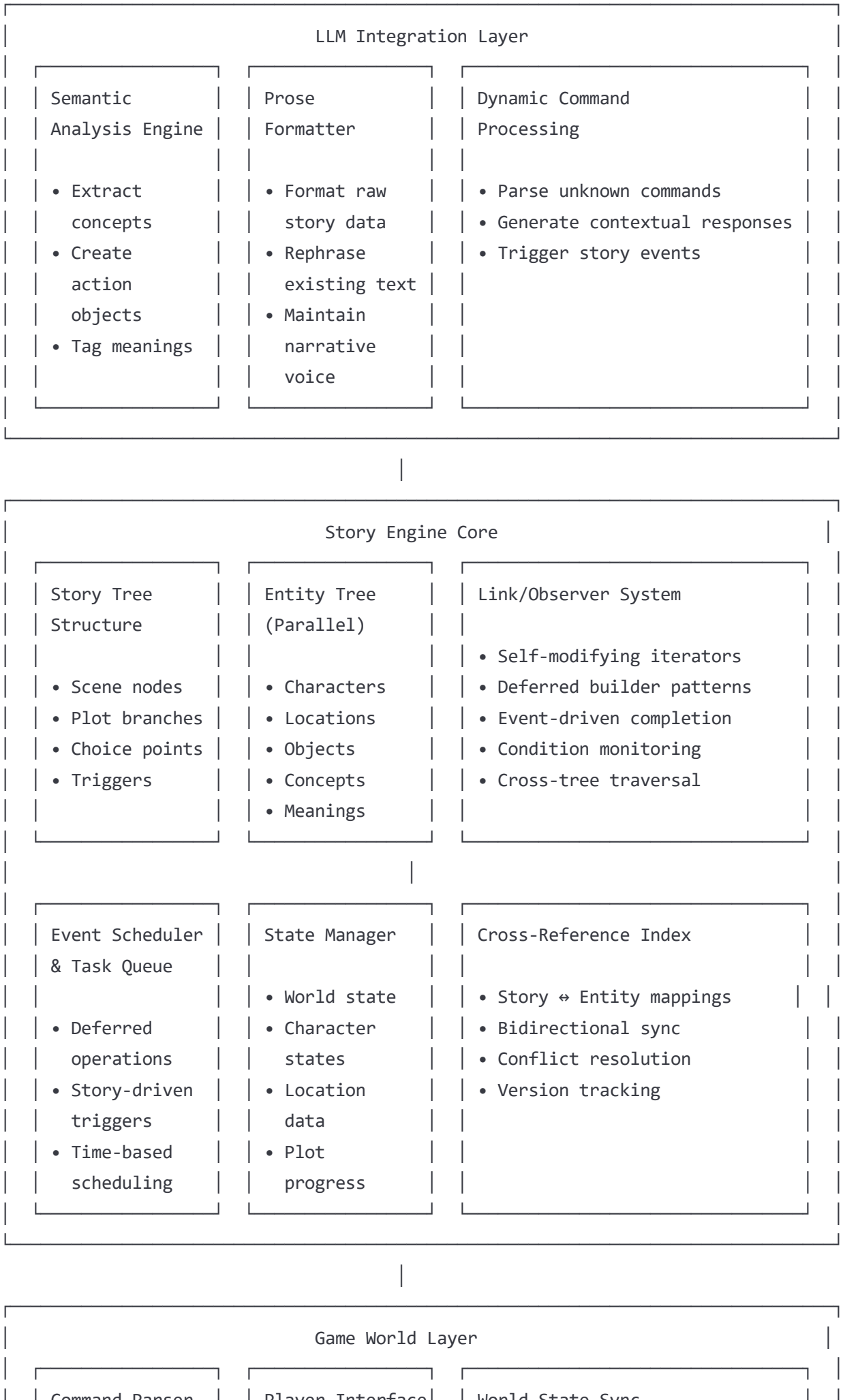
Complete Self-Modifying Story Engine Architecture

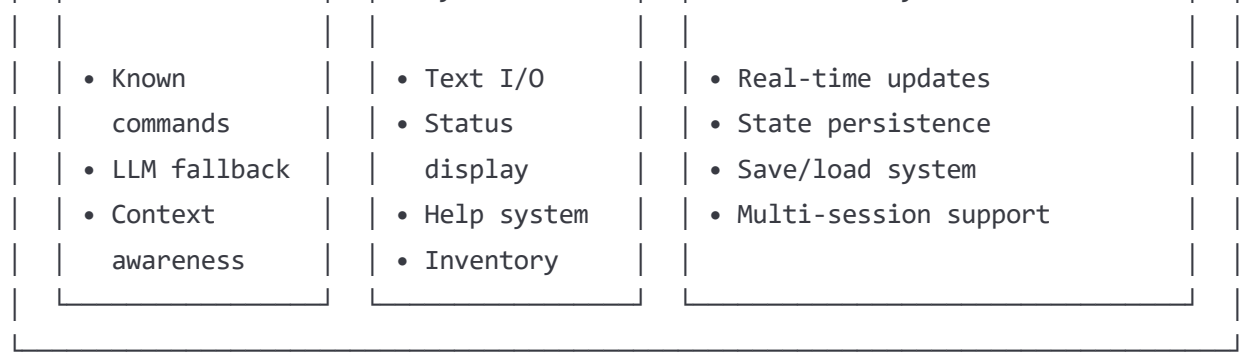
Executive Summary

This system creates an interactive "Discovery of Witches" MUD that transforms a static book into a living, explorable world where players experience the story as participants. The engine uses self-modifying iterators, deferred builder patterns, semantic analysis, and LLM integration to create dynamic narratives that adapt in real-time.

Key Innovation: Builder pattern chains that can suspend mid-execution (like `.doThis().dontForgetTo().fertilizeTree()`) where the final methods only execute when story conditions are met (arriving at Garden of Eden with money found in car).

System Overview


LLM Integration Layer
Semantic Analysis Engine
• Extract concepts
• Create action objects
• Tag meanings
Prose Formatter
• Format raw story data
• Rephrase existing text
• Maintain narrative voice
Dynamic Command Processing
• Parse unknown commands
• Generate contextual responses
• Trigger story events
Story Engine Core
Story Tree Structure
• Scene nodes
• Plot branches
• Choice points
• Triggers
Entity Tree (Parallel)
• Characters
• Locations
• Objects
• Concepts
• Meanings
Link/Observer System
• Self-modifying iterators
• Deferred builder patterns
• Event-driven completion
• Condition monitoring
• Cross-tree traversal
Event Scheduler & Task Queue
• Deferred operations
• Story-driven triggers
• Time-based scheduling
State Manager
• World state
• Character states
• Location data
• Plot progress
Cross-Reference Index
• Story ⇔ Entity mappings
• Bidirectional sync
• Conflict resolution
• Version tracking
Game World Layer
Command Parser
Player Interface
World State Sync



---

1. Story Tree & Entity Tree Architecture

Story Tree Structure

python

```python
class StoryNode:
    def __init__(self, scene_id, data):
        self.scene_id = scene_id
        self.raw_text = data.get('raw_text', '')
        self.formatted_prose = None  # Generated by LLM
        self.choices = data.get('choices', [])
        self.prerequisites = data.get('prerequisites', [])
        self.triggers = data.get('triggers', [])
        self.children = {}
        self.metadata = data.get('metadata', {})

        # Link system integration
        self.observers = []
        self.deferred_operations = []

    def add_deferred_operation(self, builder_chain, conditions):
        """Store partial builder execution for later completion"""
        self.deferred_operations.append({
            'builder': builder_chain,
            'conditions': conditions,
            'created_at': time.time()
        })
```

Entity Tree Structure

python

```python
class EntityNode:
    def __init__(self, entity_id, entity_type, data):
        self.entity_id = entity_id
        self.entity_type = entity_type  # 'character', 'location', 'object', 'concept'
        self.raw_data = data
        self.semantic_tags = []
        self.meaning_extracts = []
        self.action_objects = []

        # Cross-references to story tree
        self.story_references = set()

    def extract_meanings(self, semantic_analyzer):
        """Use LLM to extract actionable meanings from entity data"""
        self.meaning_extracts = semantic_analyzer.extract_meanings(self.raw_data)
        self.action_objects = semantic_analyzer.create_action_objects(self.meaning_extracts)
```

---

2. Self-Modifying Iterator & Link System

Core Link Class

python

```python
class LinkTraverser:
    def __init__(self, story_tree, entity_tree):
        self.story_tree = story_tree
        self.entity_tree = entity_tree
        self.active_links = []
        self.event_queue = asyncio.Queue()
        self.condition_monitors = {}

    async def traverse_with_conditions(self, start_node, conditions):
        """Self-modifying iterator that changes behavior based on conditions"""
        current = start_node
        path_history = []

        while current:
            # Check if any deferred operations can now complete
            await self._check_deferred_completions(current)

            # Create self-modifying iterator for current node
            node_iterator = self._create_adaptive_iterator(current)

            async for event in node_iterator:
                # Process event and potentially modify traversal
                if await self._should_modify_path(event):
                    current = await self._calculate_new_path(event, current)
                    break

                yield event

    def _create_adaptive_iterator(self, node):
        """Create iterator that can modify itself during traversal"""
        return AdaptiveNodeIterator(node, self.condition_monitors)

class AdaptiveNodeIterator:
    def __init__(self, node, condition_monitors):
        self.node = node
        self.condition_monitors = condition_monitors
        self.behavior_modifiers = []

    def __aiter__(self):
        return self

    async def __anext__(self):
        # Check for behavior modifications
        for modifier in self.behavior_modifiers:
            if await modifier.should_activate():
```

```
            if await modifier.should_activate():
                await modifier.apply_to_iterator(self)

        # Continue with modified or original behavior
        return await self._generate_next_event()
```

Deferred Builder Pattern

python

```python
class DeferredStoryBuilder:
    def __init__(self):
        self._operations = []
        self._execution_state = 'building'
        self._suspension_point = None
        self._completion_conditions = []

    def doThis(self, action):
        """First part of builder chain - executes immediately"""
        self._operations.append(('do_this', action))
        return self

    def thenThis(self, action):
        """Second part - executes immediately"""
        self._operations.append(('then_this', action))
        return self

    def dontForgetTo(self, action, completion_conditions):
        """Third part - creates suspension point"""
        self._operations.append(('dont_forget', action))
        self._completion_conditions = completion_conditions
        self._execution_state = 'suspended'
        return DeferredCompletion(self, action, completion_conditions)

class DeferredCompletion:
    def __init__(self, builder, action, conditions):
        self.builder = builder
        self.action = action
        self.conditions = conditions

    async def fertilizeTree(self):
        """Final method that only executes when conditions are met"""
        # Wait for conditions to be satisfied
        await self._wait_for_conditions()

        # Execute the deferred action
        return await self._execute_final_action()

    async def _wait_for_conditions(self):
        """Wait for story events like 'at_garden_of_eden' and 'has_money'"""
        condition_future = asyncio.Future()

        # Register with story event system
        story_event_manager.register_condition_waiter(
            self.conditions, condition_future
```

```
            self.conditions, condition_future
        )

        await condition_future
```

---

3. LLM Integration & Prose Formatting

Semantic Analysis Engine

python

```python
class SemanticAnalysisEngine:
    def __init__(self, llm_client):
        self.llm_client = llm_client
        self.concept_database = ConceptDatabase()

    async def extract_meanings_from_text(self, text):
        """Extract actionable concepts from raw story text"""
        prompt = f"""
        Analyze this text from Discovery of Witches and extract:
        1. Key concepts that could trigger story events
        2. Character motivations and emotional states
        3. Location descriptions and atmosphere
        4. Objects and their significance
        5. Relationships between entities

        Text: {text}

        Return as structured JSON with semantic tags.
        """

        response = await self.llm_client.chat.completions.create(
            model="gpt-4",
            messages=[{"role": "user", "content": prompt}]
        )

        return self._parse_semantic_response(response.choices[0].message.content)

    def create_action_objects(self, meanings):
        """Convert extracted meanings into executable action objects"""
        action_objects = []

        for meaning in meanings:
            if meaning['type'] == 'trigger_event':
                action_obj = TriggerActionObject(
                    conditions=meaning['conditions'],
                    actions=meaning['actions'],
                    side_effects=meaning['side_effects']
                )
                action_objects.append(action_obj)

        return action_objects

class TriggerActionObject:
    def __init__(self, conditions, actions, side_effects):
```

```
        self.conditions = conditions
        self.actions = actions
        self.side_effects = side_effects

    async def execute_if_conditions_met(self, world_state):
        """Execute action if conditions are satisfied"""
        if self._check_conditions(world_state):
            results = await self._execute_actions(world_state)
            await self._apply_side_effects(world_state, results)
            return results
        return None
```

Prose Formatting System

python

```python
class ProseFormatter:
    def __init__(self, llm_client):
        self.llm_client = llm_client
        self.style_context = {}

    async def format_story_elements(self, raw_elements, context):
        """Transform raw story data into flowing narrative prose"""
        prompt = f"""
        Take these story elements and weave them into flowing narrative prose
        in the style of Discovery of Witches:

        Story Elements: {raw_elements}
        Current Context: {context}
        Character State: {context.get('character_state', {})}
        Location: {context.get('location', 'unknown')}

        Maintain the magical, academic atmosphere of the original book.
        Write in present tense, second person for player actions.
        Include sensory details and emotional undertones.
        """

        response = await self.llm_client.chat.completions.create(
            model="gpt-4",
            messages=[{"role": "user", "content": prompt}]
        )

        return response.choices[0].message.content

    async def rephrase_existing_text(self, original_text, new_context):
        """Rephrase existing prose based on changed story context"""
        prompt = f"""
        Rephrase this existing narrative text to reflect new story context:

        Original Text: {original_text}
        New Context: {new_context}

        Maintain narrative consistency and voice while incorporating the changes.
        """

        response = await self.llm_client.chat.completions.create(
            model="gpt-4",
            messages=[{"role": "user", "content": prompt}]
        )

        return response.choices[0].message.content
```

```
        return response.choices[0].message.content
```

---

4. Event System & Story Progression

Story Event Manager

python

```python
class StoryEventManager:
    def __init__(self):
        self.event_listeners = defaultdict(list)
        self.deferred_operations = {}
        self.condition_waiters = defaultdict(list)
        self.world_state = WorldState()

    async def trigger_story_event(self, event_name, event_data):
        """Trigger story event and process all waiting operations"""
        # Update world state
        self.world_state.update(event_name, event_data)

        # Check condition waiters
        await self._check_condition_waiters(event_name, event_data)

        # Notify event listeners
        for listener in self.event_listeners[event_name]:
            await listener(event_data)

        # Process deferred operations
        await self._process_deferred_operations(event_name, event_data)

    async def _check_condition_waiters(self, event_name, event_data):
        """Check if any deferred operations can now complete"""
        completed_conditions = []

        for condition_set, waiters in self.condition_waiters.items():
            if self._conditions_satisfied(condition_set, event_name, event_data):
                for waiter in waiters:
                    waiter.set_result(event_data)
                completed_conditions.append(condition_set)

        # Clean up completed conditions
        for condition_set in completed_conditions:
            del self.condition_waiters[condition_set]

    def _conditions_satisfied(self, condition_set, event_name, event_data):
        """Check if all conditions in set are satisfied"""
        # Example: ['at_garden_of_eden', 'has_money']
        for condition in condition_set:
            if not self.world_state.check_condition(condition):
                return False
        return True

class WorldState:
```

```
class WorldState:
    def __init__(self):
        self.character_location = None
        self.character_inventory = set()
        self.character_relationships = {}
        self.plot_flags = set()
        self.time_of_day = 'morning'
        self.magical_awareness = False

    def check_condition(self, condition):
        """Check if a specific condition is met"""
        if condition == 'at_garden_of_eden':
            return self.character_location == 'garden_of_eden'
        elif condition == 'has_money':
            return 'money' in self.character_inventory
        elif condition == 'found_money_in_car':
            return 'found_money_car' in self.plot_flags
        # Add more conditions as needed
        return False
```

---

5. Game Interface & Command Processing

Dynamic Command Parser

python

```python
class DynamicCommandParser:
    def __init__(self, llm_client, story_engine):
        self.llm_client = llm_client
        self.story_engine = story_engine
        self.known_commands = {
            'look', 'examine', 'go', 'take', 'drop', 'inventory',
            'talk', 'read', 'open', 'close', 'use'
        }

    async def parse_command(self, user_input, context):
        """Parse user command with LLM fallback for unknown commands"""
        tokens = user_input.lower().split()

        if not tokens:
            return None

        command = tokens[0]

        if command in self.known_commands:
            return await self._handle_known_command(tokens, context)
        else:
            return await self._handle_unknown_command(user_input, context)

    async def _handle_unknown_command(self, user_input, context):
        """Use LLM to interpret unknown commands and generate actions"""
        prompt = f"""
        The player typed: "{user_input}"

        Current Context:
        - Location: {context.get('location', 'unknown')}
        - Available objects: {context.get('objects', [])}
        - Character state: {context.get('character_state', {})}

        Interpret this command and return:
        1. What action the player wants to take
        2. What objects/characters are involved
        3. What story events this might trigger
        4. Appropriate response text

        Respond in the style of Discovery of Witches.
        """

        response = await self.llm_client.chat.completions.create(
            model="gpt-4",
            messages=[{"role": "user", "content": prompt}]
```

```
        messages=[{"role": "user", "content": prompt}]
    )

    interpretation = self._parse_llm_response(response.choices[0].message.content)

    # Potentially trigger story events based on interpretation
    if interpretation.get('story_events'):
        for event in interpretation['story_events']:
            await self.story_engine.trigger_event(event['name'], event['data'])

    return interpretation
```

Player Interface

python

```python
class PlayerInterface:
    def __init__(self, story_engine, command_parser):
        self.story_engine = story_engine
        self.command_parser = command_parser
        self.output_buffer = []

    async def game_loop(self):
        """Main game interaction loop"""
        await self._display_intro()

        while True:
            # Display current scene
            await self._display_current_scene()

            # Get player input
            user_input = await self._get_user_input()

            if user_input.lower() in ['quit', 'exit']:
                break

            # Process command
            result = await self.command_parser.parse_command(
                user_input,
                self.story_engine.get_current_context()
            )

            # Display result
            if result:
                await self._display_result(result)

            # Check for triggered story events
            await self._process_pending_events()

    async def _display_current_scene(self):
        """Display current scene with formatted prose"""
        current_node = self.story_engine.get_current_node()

        if not current_node.formatted_prose:
            # Generate formatted prose using LLM
            current_node.formatted_prose = await self.story_engine.format_scene_prose(current_r

        print(current_node.formatted_prose)

        # Display available actions
        if current_node.choices:
```

```
    if current_node.choices:
        print("\nAvailable actions:")
        for i, choice in enumerate(current_node.choices, 1):
            print(f"{i}. {choice}")
```

---

6. Complete Integration Example

Discovery of Witches Implementation

python

```python
class DiscoveryOfWitchesEngine:
    def __init__(self):
        self.story_tree = self._build_story_tree()
        self.entity_tree = self._build_entity_tree()
        self.event_manager = StoryEventManager()
        self.semantic_analyzer = SemanticAnalysisEngine(llm_client)
        self.prose_formatter = ProseFormatter(llm_client)
        self.link_traverser = LinkTraverser(self.story_tree, self.entity_tree)

        # Initialize with Bodleian Library scene
        self.current_node = self.story_tree.get_node('bodleian_library')

    async def setup_garden_of_eden_scenario(self):
        """Example of deferred builder pattern implementation"""

        # Character needs to get to Garden of Eden and can't forget to fertilize tree
        # But they need money first (found in car)
        garden_quest = (DeferredStoryBuilder()
                        .doThis("travel_towards_garden")  # Executes immediately
                        .thenThis("search_for_supplies")  # Executes immediately
                        .dontForgetTo("fertilize_tree_of_life",
                                      conditions=['at_garden_of_eden', 'has_money'])
                        .fertilizeTree())  # Only executes when conditions met

        # Register the deferred operation with event system
        await self.event_manager.register_deferred_operation(
            'garden_quest', garden_quest
        )

        # Set up the money search trigger
        car_search = (DeferredStoryBuilder()
                      .doThis("arrive_at_gas_station")
                      .thenThis("realize_missing_money")
                      .dontForgetTo("search_car_for_money",
                                    conditions=['player_searches_car'])
                      .findMoney())

        await self.event_manager.register_deferred_operation(
            'money_search', car_search
        )

    async def handle_player_command(self, command):
        """Process player command with full system integration"""

        # Parse command (known or LLM-interpreted)
```

```
        # Parse command (known or LLM-interpreted)
        parsed = await self.command_parser.parse_command(
            command, self.get_current_context()
        )

        # Extract semantic meaning
        meanings = await self.semantic_analyzer.extract_meanings_from_text(
            parsed.get('action_description', '')
        )

        # Create action objects
        action_objects = self.semantic_analyzer.create_action_objects(meanings)

        # Execute actions and check for story events
        for action_obj in action_objects:
            result = await action_obj.execute_if_conditions_met(
                self.event_manager.world_state
            )

            if result:
                await self.event_manager.trigger_story_event(
                    result['event_name'], result['event_data']
                )

        # Format response prose
        response_text = await self.prose_formatter.format_story_elements(
            parsed, self.get_current_context()
        )

        return response_text
```

Implementation Checklist

Phase 1: Core Infrastructure ✅ TO DO

- [ ] **Story Tree Implementation**
  - Basic node structure with metadata
  - Tree traversal algorithms
  - Save/load functionality
- [ ] **Entity Tree Implementation**
  - Parallel structure to story tree
  - Cross-referencing system
  - Bidirectional synchronization
- [ ] **Link/Observer System**
  - Basic observer pattern
  - Event registration/notification
  - Condition monitoring framework

Phase 2: Advanced Patterns ⚠️ COMPLEX

- [ ] **Self-Modifying Iterators**
  - Coroutine-based suspension
  - Generator state preservation
  - Context variable management
- [ ] **Deferred Builder Pattern**
  - Partial execution tracking
  - Condition-based completion
  - State serialization/restoration
- [ ] **Event-Driven Architecture**
  - Asyncio event loop integration
  - Priority-based scheduling
  - Story-event triggers

Phase 3: LLM Integration 🤖 CRITICAL

- [ ] **Semantic Analysis Engine**
  - Text-to-meaning extraction
  - Concept tagging database
  - Action object generation
- [ ] **Prose Formatting System**
  - Raw data to narrative prose
  - Style consistency maintenance
  - Context-aware rephrasing
- [ ] **Dynamic Command Processing**
  - Known command handling
  - LLM fallback for unknown commands
  - Context-aware interpretation

Phase 4: Game Interface 🎮 USER-FACING

- [ ] **Command Parser**
  - Text input processing
  - Command routing
  - Help system
- [ ] **Player Interface**
  - Text output formatting
  - Status display
  - Inventory management
- [ ] **World State Management**
  - Real-time state updates
  - Persistence layer
  - Multi-session support

Phase 5: Discovery of Witches Content 📚 CONTENT

- [ ] **Book Analysis & Parsing**
  - Scene extraction from text
  - Character relationship mapping
  - Location descriptions
- [ ] **Story Tree Population**
  - Scene node creation
  - Choice point definition
  - Trigger condition setup
- [ ] **Entity Tree Population**
  - Character entity creation
  - Location entity setup
  - Object and concept mapping

Critical Dependencies

Technical Requirements

- **Python 3.9+** (for advanced asyncio features)
- **OpenAI API** or equivalent LLM service
- **PostgreSQL** (for semantic relationship storage)
- **Redis** (for real-time event caching)
- **Docker** (for deployment consistency)

Key Libraries

- `asyncio` - Coroutine and event loop management
- `aiohttp` - Async HTTP client for LLM API calls
- `sqlalchemy` - Database ORM for story/entity data
- `redis-py` - Event caching and pub/sub
- `dill` - Advanced object serialization
- `spacy` - Text preprocessing for semantic analysis
- `transformers` - Local LLM models (optional)

Development Challenges

Highest Risk Items:

1. **Deferred Builder State Management** - Complex serialization/restoration
2. **LLM Integration Reliability** - API rate limits and failure handling
3. **Real-time Event Coordination** - Race conditions in story progression
4. **Memory Management** - Long-lived suspended operations
5. **Story Content Creation** - Manual scene/entity extraction from book

Success Dependencies:

- Robust error handling for LLM API failures
- Efficient state serialization for complex builder chains
- Clear separation between story logic and presentation
- Comprehensive testing of deferred operation scenarios
- Performance optimization for real-time prose generation

This system represents a novel approach to interactive storytelling that bridges static literature and dynamic gaming through advanced Python patterns and LLM integration.