Baseline

AgreeMate System Architecture & Data Flow

1. Directory Structure

```
baseline/
├── utils/
│   ├── data_loader.py        # Raw CSV data handling
│   └── model_loader.py       # HuggingFace model management
├── agents/
│   ├── base_agent.py         # Core negotiation behavior
│   ├── buyer.py              # Buyer-specific logic
│   └── seller.py             # Seller-specific logic
├── config.py                 # System configuration
├── strategies.py             # Strategy definitions
├── dspy_manager.py           # DSPy LM configuration
├── scenario_manager.py       # Dataset management
├── negotiation_runner.py     # Negotiation execution
├── experiment_state.py       # Progress tracking
├── metrics_collector.py      # Analysis tools
├── experiment_runner.py      # Main orchestrator
└── run_experiments.py        # CLI entry point
```

2. Core Components

2.1 Configuration (config.py)

- Defines experiment parameters and model configurations
- Handles validation of all configurable parameters
- Contains pre-defined experiment templates:
  - Baseline: Single model evaluation
  - Model Comparison: Cross-model analysis
  - Strategy Analysis: Strategy effectiveness study

2.2 Data Management Layer

- **DataLoader**:
  - Handles raw CSV parsing
  - Maintains data splits (train/test/val)
  - Performs initial data validation
- **ScenarioManager**:
  - Creates negotiation scenarios
  - Manages category balancing
  - Validates price relationships

- Provides context to agents

2.3 Model Management Layer

- **ModelLoader**:
  - Handles raw HuggingFace model loading
  - Manages model caching
  - Handles multi-GPU allocation
- **DSPyManager**:
  - Configures DSPy LMs for negotiation
  - Handles strategy-specific adjustments
  - Manages parallel execution contexts

2.4 Negotiation Layer

- **BaseAgent**:
  - Implements core negotiation logic
  - Manages state tracking
  - Handles message generation
- **Buyer/Seller Agents**:
  - Implement role-specific behaviors
  - Handle price boundaries
  - Track utility metrics

2.5 Execution Layer

- **NegotiationRunner**:
  - Manages individual negotiations
  - Handles turn-taking
  - Validates price movements
- **ExperimentRunner**:
  - Orchestrates full experiments
  - Manages parallel execution
  - Handles resource allocation

2.6 Analysis Layer

- **MetricsCollector**:
  - Tracks negotiation metrics
  - Computes statistics
  - Generates reports
- **ExperimentState**:
  - Manages checkpointing

- Handles experiment recovery
- Tracks progress

3. Data Flow

3.1 Initialization Flow

1. Load configuration
2. Initialize components
3. Setup logging and output directories
4. Validate experiment parameters

3.2 Scenario Creation Flow

1. Load raw CSV data
2. Create negotiation scenarios
3. Balance categories
4. Validate price relationships
5. Provide context to agents

3.3 Negotiation Flow

1. Initialize agent pair
2. Configure DSPy LMs
3. Execute turns
4. Track metrics
5. Handle completion

3.4 Results Flow

1. Collect metrics
2. Compute statistics
3. Generate reports
4. Save checkpoints
5. Export analysis

4. Key Interfaces

4.1 Agent Interface

```
class BaseAgent:
    async def step(self) -> Dict:
        """Generate next negotiation move."""
        pass

    def compute_utility(self, price: float) -> float:
        """Compute agent's utility for given price."""
```

```
        pass

    def get_strategy_adherence(self) -> float:
        """Measure how well agent follows strategy."""
        pass
```

4.2 Message Format

```
{
    'role': 'buyer/seller',      # Agent role
    'content': str,              # Natural language message
    'price': Optional[float],    # Current offer
    'status': str                # offer/counter/accept/reject
}
```

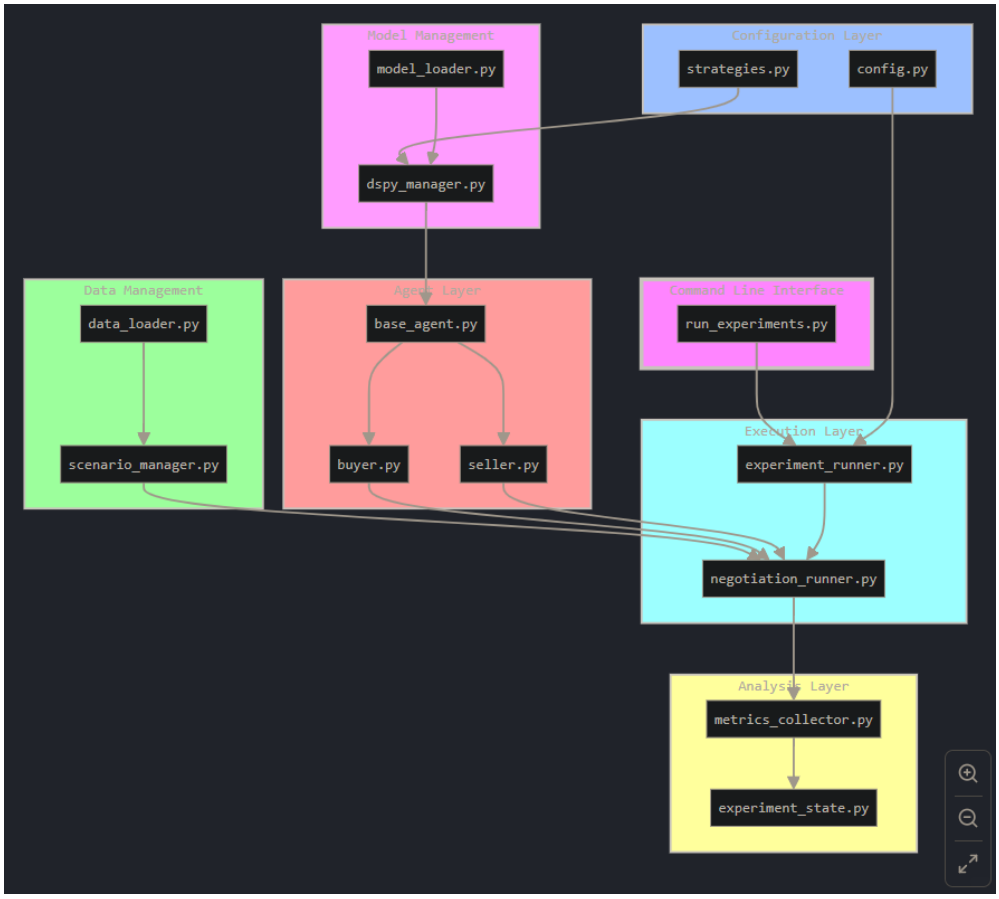
5. Configuration Options

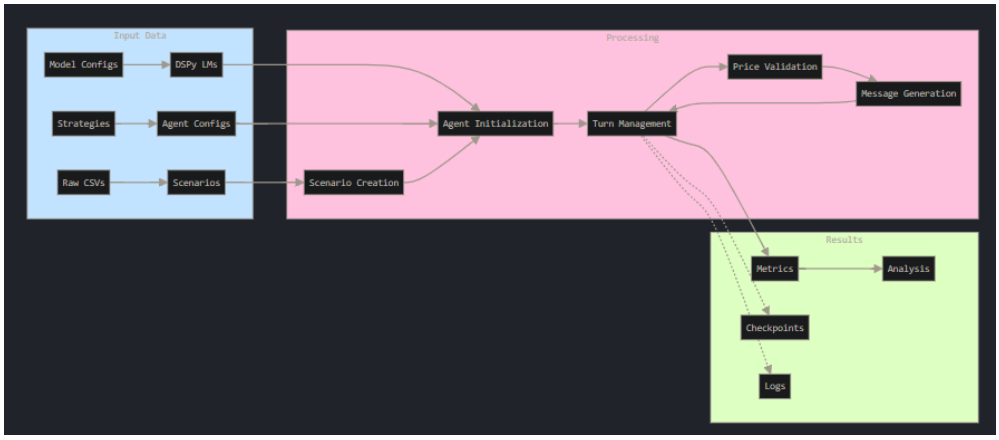
5.1 Model Configuration

```
@dataclass
class ModelConfig:
    name: str          # HuggingFace model name
    max_tokens: int    # Maximum context length
    temperature: float # Generation temperature
    prompt_template: str # Template for generation
```

5.2 Experiment Configuration

```
@dataclass
class ExperimentConfig:
    num_scenarios: int   # Number of scenarios to run
    max_turns: int      # Maximum turns per negotiation
    turn_timeout: float # Seconds per turn
    models: List[str]   # Models to use
    strategies: List[str] # Strategies to test
```





Usage Guide

Quick Start

```
# Run baseline experiment
python -m baseline.run_experiments \\
    --output ./experiments \\
    --config baseline \\
    --name my_baseline_001
```

```
# Run model comparison with debug logging
python -m baseline.run_experiments \\
    --output ./experiments \\
    --config model_comparison \\
    --name model_comp_001 \\
    --debug
```

Running Experiments

1. Setup Environment

```
# Clone repository
git clone <https://github.com/your-org/agreemate.git>
cd agreemate

# Install dependencies
pip install -r requirements.txt

# Set up model cache directory (optional)
export DSPY_CACHE_DIR=/path/to/cache
```

2. Configure Experiment

Use predefined configurations or create custom ones in `config.py`:

```
# Example custom configuration
EXPERIMENT_CONFIGS["my_config"] = ExperimentConfig(
    num_scenarios=50,
    max_turns=15,
    turn_timeout=30.0,
    models=["llama-3.1-8b"],
    strategies=["cooperative", "fair"]
)
```

3. Monitor Progress

```
# Check experiment logs
tail -f experiments/my_baseline_001/logs/experiment.log

# Monitor results directory
ls -l experiments/my_baseline_001/results/

# Check latest checkpoint
ls -t experiments/my_baseline_001/checkpoints/ | head -1
```

4. Analyze Results

```
# Using provided analysis tools
from baseline.metrics_collector import MetricsCollector

# Load results
results_path = "experiments/my_baseline_001/results/my_baseline_001_results.json"
collector = MetricsCollector()
analysis = collector.analyze_results(results_path)

# Generate reports
collector.export_results(analysis, format='all')
```

Common Configurations

1. Baseline Generation

Best for initial testing and baseline establishment:

```
python -m baseline.run_experiments \\
    --output ./experiments \\
    --config baseline
```

2. Model Comparison

For comparing different model sizes/architectures:

```
python -m baseline.run_experiments \\
    --output ./experiments \\
    --config model_comparison
```

3. Strategy Analysis

For deep-diving into negotiation strategies:

```
python -m baseline.run_experiments \\
    --output ./experiments \\
    --config strategy_analysis
```

Output Structure

```
experiments/
└── {experiment_name}_{timestamp}/
    ├── results/
    │   ├── summary.csv          # High-level metrics
    │   ├── negotiations.json    # Detailed results
    │   ├── model_analysis.csv   # Model comparisons
    │   └── strategy_analysis.csv # Strategy metrics
    ├── checkpoints/
    │   └── checkpoint_{timestamp}.json
```

```
    └── logs/
        └── experiment.log
```

AgreeMate Data Handling & Scenarios

1. Dataset Structure

1.1 Raw Data Format

```
scenario_id,split_type,category,list_price,buyer_target,seller_target,title,description,price_delta_pct,relative_price,title_token_count,description_length,data_completeness,price_confidence,has_images
train_00000,train,electronics,10.0,7.0,10.0,Product Title,Description Text,0.3,0.1,11,520,1.0,True,True
```

1.2 Core Fields

- **Identifiers**
  - `scenario_id`: Unique identifier (e.g., 'train_00000')
  - `split_type`: Dataset split ('train', 'test', 'validation')
  - `category`: Item category ('electronics', 'vehicles', 'furniture', 'housing')
- **Price Information**
  - `list_price`: Original listing price
  - `buyer_target`: Buyer's target price
  - `seller_target`: Seller's target price
  - `price_delta_pct`: Percentage difference between targets
  - `relative_price`: Price relative to category median
- **Item Details**
  - `title`: Item title
  - `description`: Item description
  - `title_token_count`: Number of tokens in title
  - `description_length`: Length of description
- **Quality Metrics**
  - `data_completeness`: Record completeness score (0-1)
  - `price_confidence`: Price validation flag
  - `has_images`: Image availability flag

2. Data Loading Pipeline

2.1 Initial Loading (DataLoader)

```
class DataLoader:
    def load_splits(self) -> Tuple[pd.DataFrame, pd.DataFrame, pd.DataFrame]:
        """Load train/test/validation splits."""
        return train_df, test_df, val_df

    def get_category_stats(self) -> Dict:
        """Get category distribution and price statistics."""
        return {
            'category': {
                'count': int,
                'price_stats': Dict[str, float],
                'avg_price_delta': float
            }
        }
```

2.2 Data Validation Rules

1. Price Relationships
   - Buyer target ≤ List price
   - Seller target ≤ List price
   - Valid price_delta_pct
2. Category Validation
   - Must be in allowed categories
   - Category-specific price ranges
   - Consistent price distributions
3. Quality Checks
   - Complete required fields
   - Valid numeric values
   - Non-empty descriptions

3. Scenario Management

3.1 Scenario Creation

```
@dataclass
class NegotiationScenario:
    """Complete negotiation scenario."""
    # Core data
    scenario_id: str
    category: str

    # Price information
    list_price: float
    buyer_target: float
    seller_target: float
```

```
    # Context
    title: str
    description: str

    # Metrics
    price_delta_pct: float
    relative_price: float

    def get_buyer_context(self) -> Dict:
        """Get buyer's view of scenario."""
        pass

    def get_seller_context(self) -> Dict:
        """Get seller's view of scenario."""
        pass
```

3.2 Scenario Selection

1. Category Balancing

```
def create_evaluation_batch(
    split: str = 'test',
    size: Optional[int] = None,
    balanced_categories: bool = True
) -> List[NegotiationScenario]:
    """Create balanced scenario batch."""
    pass
```

2. Price Range Distribution
   - Low tier: $0-3,000
   - Mid tier: $3,000-10,000
   - High tier: $10,000+
3. Selection Criteria
   - Category distribution
   - Price range coverage
   - Quality thresholds
   - Complete metadata

4. Context Generation

4.1 Buyer Context

```
{
    'role': 'buyer',
    'scenario_id': str,
    'category': str,
    'item': {
        'title': str,
```

```
        'description': str,
        'list_price': float
    },
    'target_price': float  # buyer_target
}
```

4.2 Seller Context

```
{
    'role': 'seller',
    'scenario_id': str,
    'category': str,
    'item': {
        'title': str,
        'description': str,
        'list_price': float
    },
    'target_price': float  # seller_target
}
```

5. Category-Specific Handling

5.1 Electronics

- Tight price margins
- Technical specifications important
- Warranty considerations

5.2 Vehicles

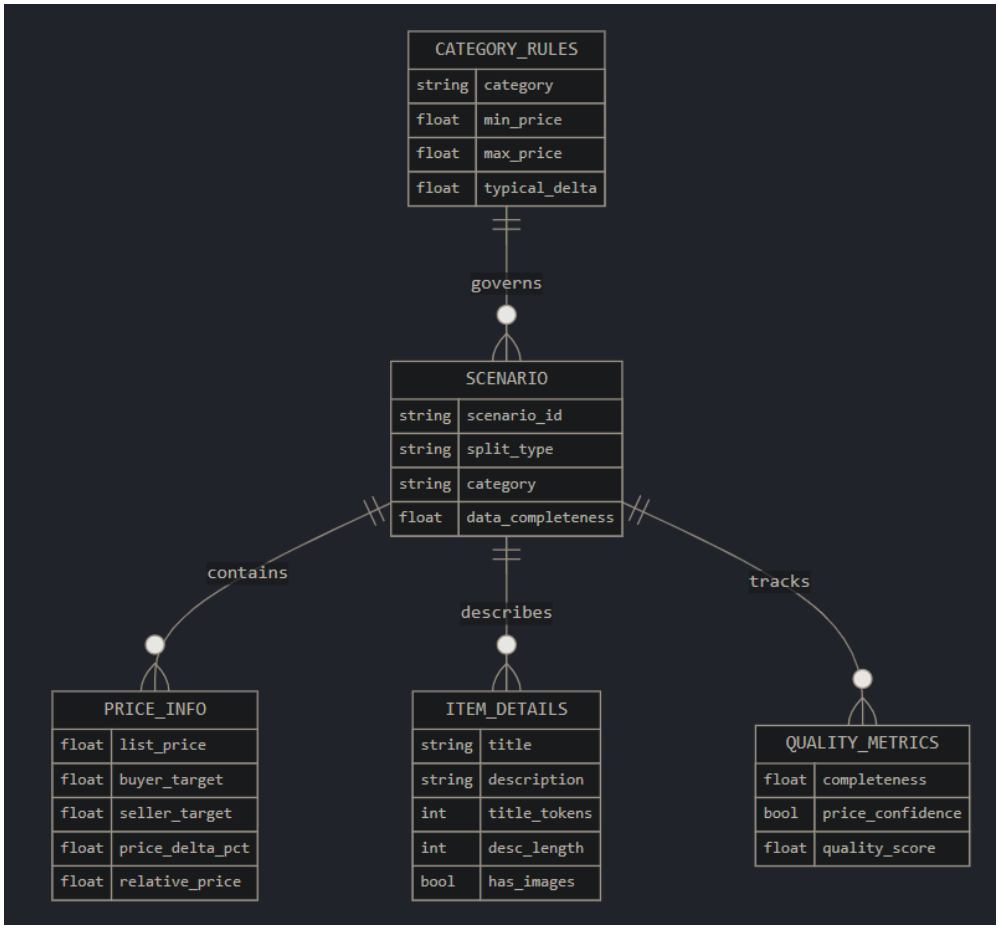
- Wide price ranges
- Condition crucial
- Multiple negotiation factors

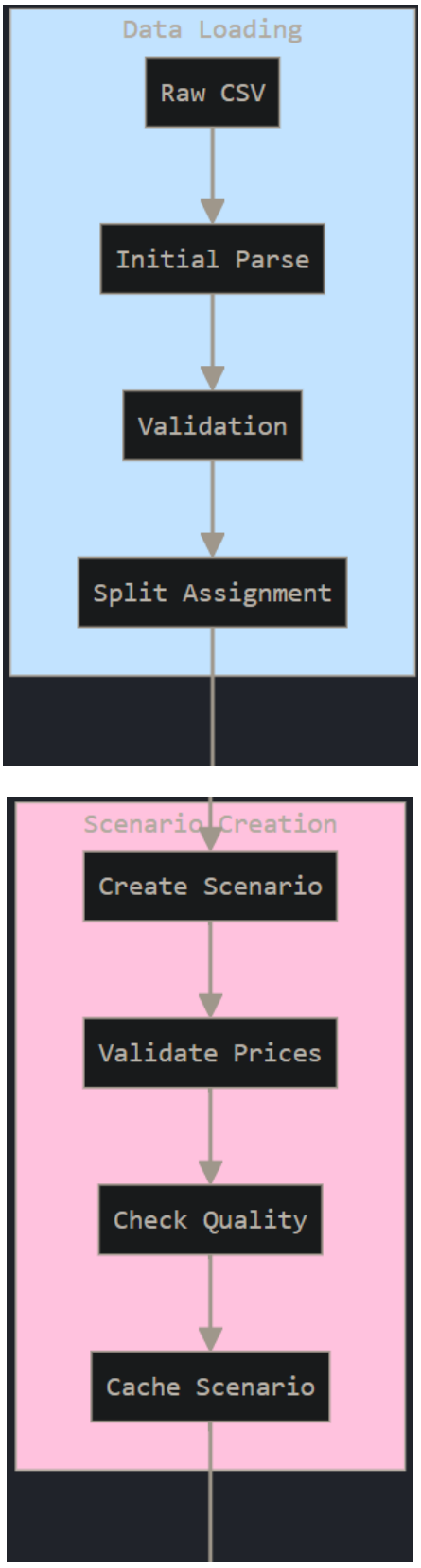
5.3 Furniture

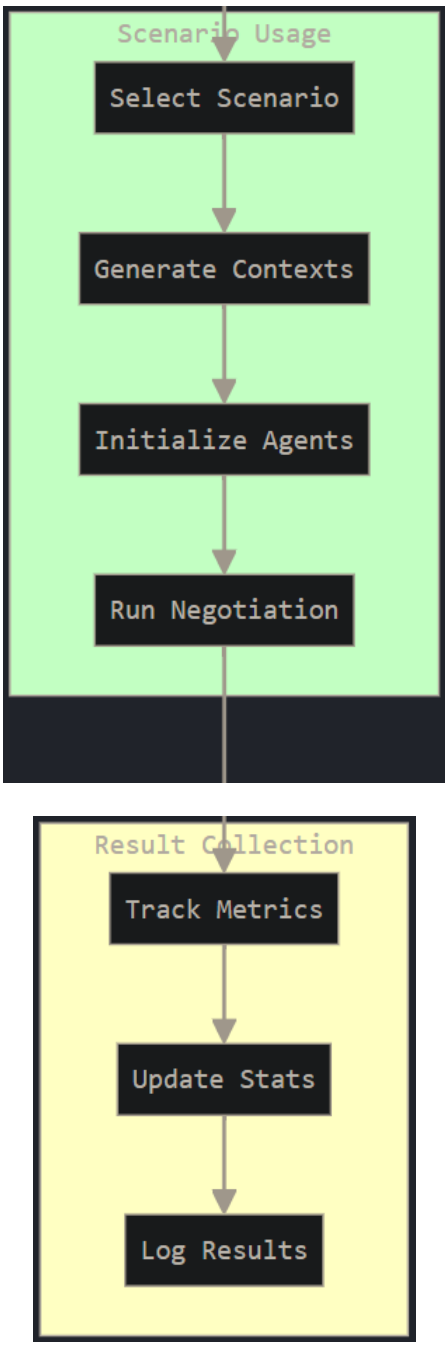
- Delivery considerations
- Condition important
- Quick turnover dynamics

5.4 Housing

- Location-based pricing
- Long-term implications
- Complex terms


CATEGORY_RULES
string category
float min_price
float max_price
float typical_delta
governs
SCENARIO
string scenario_id
string split_type
string category
float data_completeness
contains
describes
tracks
PRICE_INFO
float list_price
float buyer_target
float seller_target
float price_delta_pct
float relative_price
ITEM_DETAILS
string title
string description
int title_tokens
int desc_length
bool has_images
QUALITY_METRICS
float completeness
bool price_confidence
float quality_score


Data Loading
Raw CSV
Initial Parse
Validation
Split Assignment
Scenario Creation
Create Scenario
Validate Prices
Check Quality
Cache Scenario



AgreeMate Negotiation Core & Agent Behavior

1. Core Strategy Framework

1.1 Strategy Definitions

```
STRATEGIES = {
    "fair": {
        "description": "Balanced negotiator seeking mutual benefit",
```

```
        "risk_tolerance": "moderate",
        "patience": "moderate",
        "initial_approach": "Start with market value",
        "communication_style": "Clear, professional"
    },
    "aggressive": {
        "description": "Maximizes own value, firm stance",
        "risk_tolerance": "high",
        "patience": "high",
        "initial_approach": "Start high/low",
        "communication_style": "Direct, confident"
    },
    "cooperative": {
        "description": "Prioritizes agreement, flexible",
        "risk_tolerance": "low",
        "patience": "low",
        "initial_approach": "Start near middle",
        "communication_style": "Warm, friendly"
    }
}
```

1.2 Strategy Components

- **Risk Tolerance**: Affects price movement size
- **Patience**: Influences turns before concession
- **Initial Approach**: First offer strategy
- **Communication Style**: Tone and presentation

2. Agent Architecture

2.1 Base Agent

```
class BaseAgent:
    def __init__(
        self,
        strategy_name: str,
        target_price: float,
        category: str,
        lm: dspy.LM
    ):
        self.strategy = STRATEGIES[strategy_name]
        self.target_price = target_price
        self.category = category
        self.lm = lm
        self.conversation_history = []
        self.price_history = []

    async def step(self) -> Dict:
        """Take negotiation turn."""
        pass
```

```
    def compute_utility(self, price: float) -> float:
        """Compute utility for given price."""
        pass
```

2.2 Buyer Agent

```
class BuyerAgent(BaseAgent):
    def __init__(
        self,
        max_price: float,  # Upper bound
        **kwargs
    ):
        super().__init__(**kwargs)
        self.max_price = max_price
        self.best_offer_seen = float('inf')

    def should_accept(self, price: float) -> bool:
        """Decide whether to accept offer."""
        if price > self.max_price:
            return False
        return self._evaluate_offer(price)
```

2.3 Seller Agent

```
class SellerAgent(BaseAgent):
    def __init__(
        self,
        min_price: float,  # Lower bound
        initial_price: float,
        **kwargs
    ):
        super().__init__(**kwargs)
        self.min_price = min_price
        self.initial_price = initial_price
        self.best_offer_seen = 0.0

    def should_accept(self, price: float) -> bool:
        """Decide whether to accept offer."""
        if price < self.min_price:
            return False
        return self._evaluate_offer(price)
```

3. Decision Making

3.1 Price Movement

```
def calculate_next_price(
    self,
```

```
    current_price: float,
    opponent_price: float
) -> float:
    """Calculate next offer price."""
    # Strategy factors
    risk_factor = {
        'high': 0.8,    # Aggressive
        'moderate': 0.5, # Fair
        'low': 0.3      # Cooperative
    }[self.strategy['risk_tolerance']]

    # Movement size
    gap = abs(current_price - opponent_price)
    move_size = gap * risk_factor

    return self._adjust_price(
        current_price,
        move_size
    )
```

3.2 Offer Evaluation

```
def _evaluate_offer(
    self,
    price: float,
    context: Dict
) -> bool:
    """
    Evaluate whether to accept an offer.
    Uses strategy, price history, and context.
    """
    # Basic bounds check
    if not self._within_bounds(price):
        return False

    # Utility threshold
    utility = self.compute_utility(price)
    if utility < self.min_utility:
        return False

    # Strategy-based factors
    urgency = self._compute_urgency()
    market_position = self._assess_market()

    return self._make_decision(
        utility, urgency, market_position
    )
```

4. Language Generation

4.1 Message Structure

```
{
    'role': str,          # 'buyer' or 'seller'
    'content': str,       # Generated message
    'price': float,       # Current offer
    'status': str,        # offer/counter/accept/reject
    'reasoning': str      # Internal justification
}
```

4.2 Prompt Template

```
PROMPT_TEMPLATE = """
You are a {role} negotiating for {item}.
Your strategy is: {strategy}

Current conversation:
{history}

Your target price is: ${target_price}
Current offer: ${current_price}

Respond as {role}:"""
```

5. Negotiation Flow

5.1 Turn Structure

1. Receive opponent's message
2. Update state (prices, history)
3. Evaluate position
4. Generate response
5. Validate move
6. Send message

5.2 State Tracking

```
class NegotiationState:
    def __init__(self):
        self.turns_taken = 0
        self.messages = []
        self.price_history = []
        self.last_action = None
        self.strategy_adherence = 1.0

    def update(self, message: Dict):
        """Update state with new message."""
        self.messages.append(message)
        if message['price']:
```

```
            self.price_history.append(message['price'])
        self.turns_taken += 1
```

6. Strategy Implementation

6.1 Price Movement Rules

- Buyer increases only
- Seller decreases only
- Movement size based on strategy
- Respect bounds always

6.2 Communication Rules

```
def format_message(
    self,
    price: float,
    action: str
) -> str:
    """Format message based on strategy."""
    templates = {
        'fair': {
            'offer': "I can offer {price}.",
            'counter': "Let me counter with {price}.",
            'accept': "I accept your offer of {price}.",
            'reject': "I cannot accept that price."
        },
        # ... other strategy templates
    }
    return templates[self.strategy['name']][action]
```

7. Utilities

7.1 Buyer Utility

```
def compute_buyer_utility(
    price: float,
    target: float,
    max_price: float
) -> float:
    """Compute buyer's utility (0-1)."""
    if price > max_price:
        return 0.0
    return 1.0 - (price - target) / (max_price - target)
```

7.2 Seller Utility

```
def compute_seller_utility(
    price: float,
```

```
    target: float,
    min_price: float
) -> float:
    """Compute seller's utility (0-1)."""
    if price < min_price:
        return 0.0
    return (price - min_price) / (target - min_price)
```

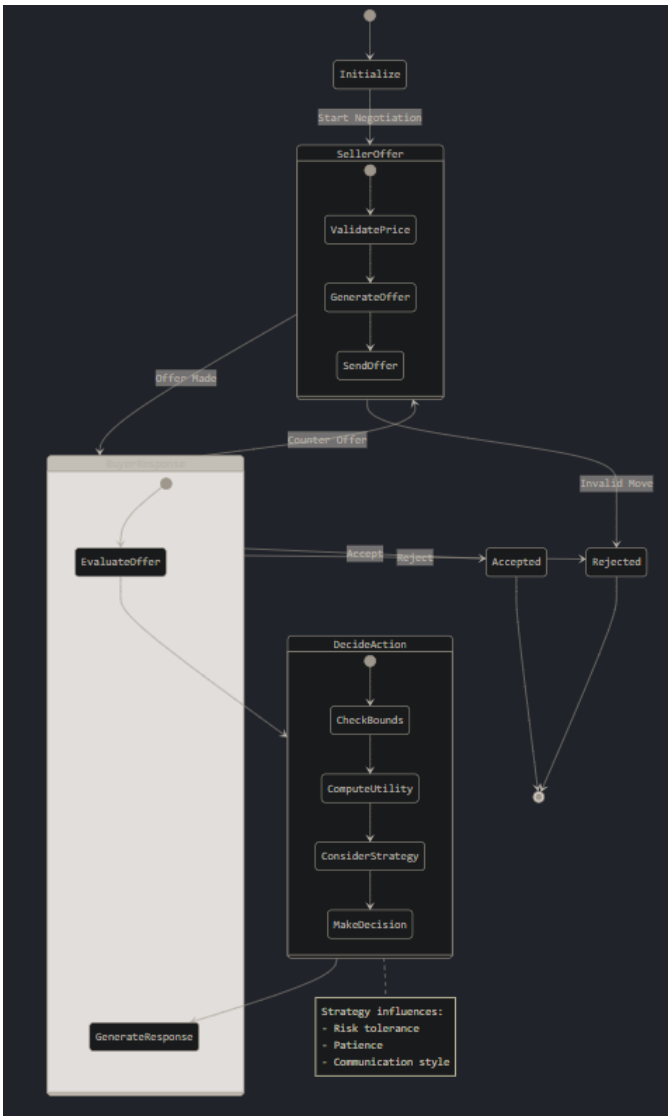
8. Error Handling

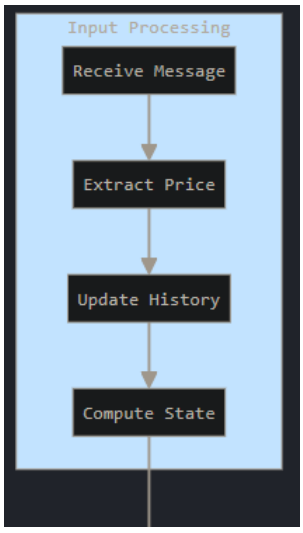
8.1 Validation

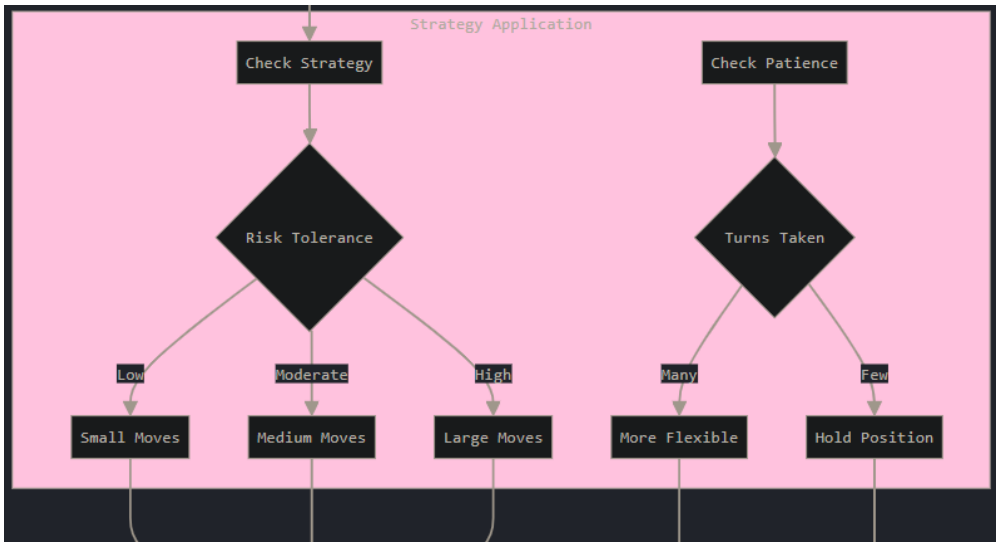
```
def validate_move(
    self,
    price: float,
    action: str
) -> bool:
    """Validate negotiation move."""
    if action == 'accept':
        return self._validate_acceptance(price)
    if action == 'reject':
        return self._validate_rejection(price)
    return self._validate_price_movement(price)
```

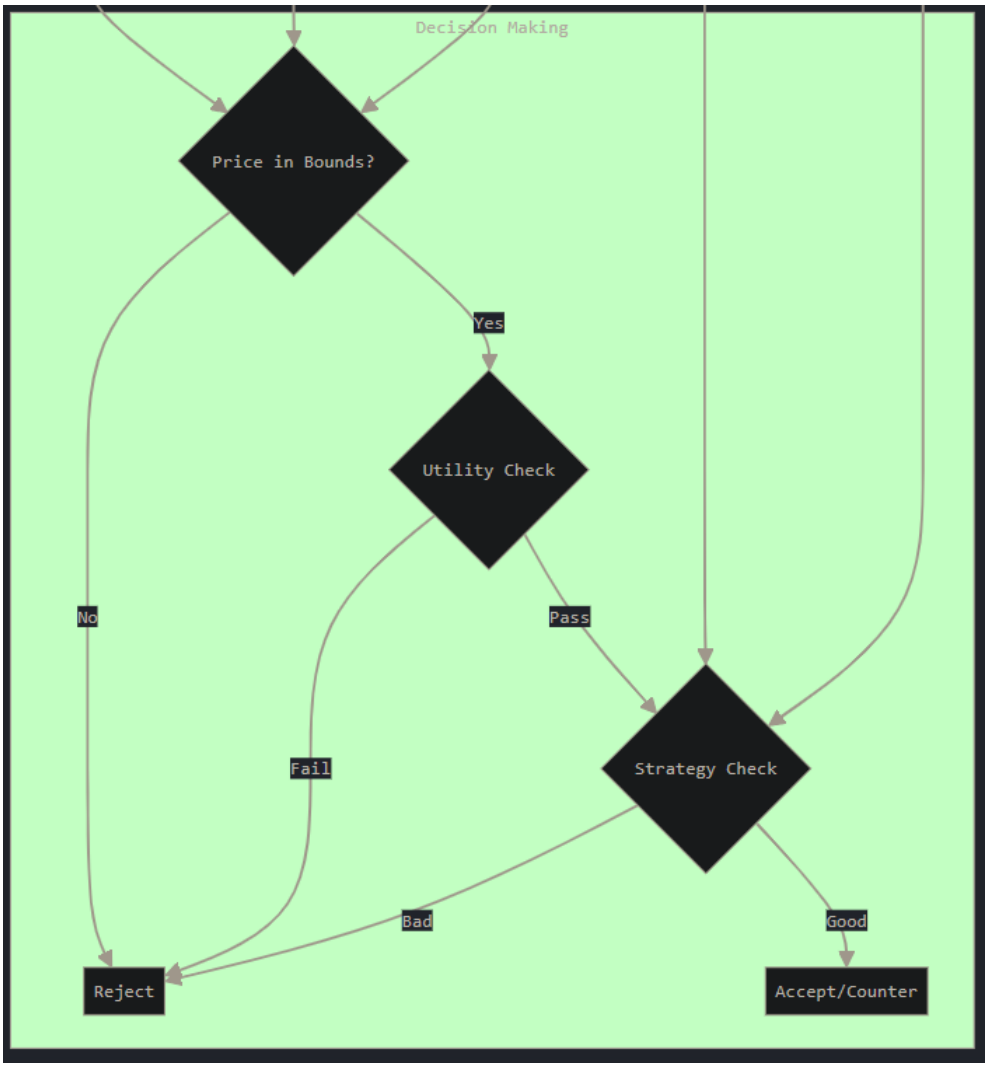
8.2 Recovery

- Timeout handling
- Invalid move recovery
- State consistency checks
- History validation


Initialize
Start Negotiation
SellerOffer
ValidatePrice
GenerateOffer
SendOffer
Offer Made
Counter Offer
Invalid Move
EvaluateOffer
Accept
Reject
Accepted
Rejected
DecideAction
CheckBounds
ComputeUtility
ConsiderStrategy
MakeDecision
GenerateResponse
Strategy influences:
- Risk tolerance
- Patience
- Communication style


Input Processing
Receive Message
Extract Price
Update History
Compute State





AgreeMate Experiment Management & Results

1. Experiment Configuration

1.1 Pre-defined Configurations

```
EXPERIMENT_CONFIGS = {
    "baseline": ExperimentConfig(
        num_scenarios=100,
        max_turns=20,
        turn_timeout=30.0,
        models=["llama-3.1-8b"],
        strategies=["cooperative", "fair", "aggressive"]
    ),
    "model_comparison": ExperimentConfig(
        num_scenarios=200,
        max_turns=20,
        turn_timeout=30.0,
        models=[
            "llama-3.1-8b",
            "llama-3.1-70b",
            "nemotron-70b"
        ],
        strategies=["cooperative", "fair"]
    ),
    "strategy_analysis": ExperimentConfig(
        num_scenarios=150,
        max_turns=25,
        turn_timeout=30.0,
        models=["llama-3.1-70b"],
        strategies=[
            "cooperative",
            "fair",
            "aggressive"
        ]
    )
}
```

1.2 Directory Structure

```
experiments/
└── {experiment_name}_{timestamp}/
    ├── results/
    │   ├── {experiment_name}_summary.csv
    │   └── {experiment_name}_results.json
    ├── checkpoints/
    │   └── checkpoint_{timestamp}.json
    └── logs/
        └── experiment.log
```

2. Metrics Collection

2.1 Core Metrics

```
@dataclass
class NegotiationMetrics:
    """Single negotiation metrics."""
    start_time: datetime
    end_time: Optional[datetime]
    turns_taken: int
    final_price: Optional[float]
    buyer_utility: Optional[float]
    seller_utility: Optional[float]
    strategy_adherence: Dict[str, float]
    messages: List[Dict]
```

2.2 Aggregate Metrics

```
@dataclass
class ModelPairMetrics:
    """Metrics for model pair combinations."""
    buyer_model: str
    seller_model: str
    num_negotiations: int
    deal_rate: float
    avg_turns: float
    avg_buyer_utility: float
    avg_seller_utility: float
    avg_duration: float
    strategy_adherence: Dict[str, float]
```

3. Experiment Execution

3.1 Main Execution Loop

```
async def run_experiment(config: ExperimentConfig):
    """Execute complete experiment."""
    # Setup
    scenarios = scenario_manager.create_evaluation_batch(
        size=config.num_scenarios
    )

    # Generate combinations
    model_combinations = _generate_model_combinations(
        config.models,
        config.strategies
    )

    # Run negotiations
    for combination in model_combinations:
        results = await _run_combination(
            combination,
```

```
            scenarios
        )
        _process_results(results)
        _save_checkpoint()
```

3.2 Progress Tracking

```
@dataclass
class ExperimentState:
    """Current experiment state."""
    experiment_name: str
    config: ExperimentConfig
    start_time: datetime
    scenarios_total: int
    scenarios_completed: int
    scenarios_failed: int
    last_checkpoint: Optional[datetime]
```

4. Results Analysis

4.1 Core Analysis

```
def analyze_results(
    completed_negotiations: Dict[str, NegotiationMetrics]
) -> Dict:
    """Analyze experiment results."""
    return {
        'deal_rates': _analyze_deal_rates(),
        'utility_analysis': _analyze_utilities(),
        'strategy_analysis': _analyze_strategies(),
        'efficiency_metrics': _analyze_efficiency()
    }
```

4.2 Strategy Analysis

```
def analyze_strategy_effectiveness(
    results: Dict[str, NegotiationMetrics]
) -> pd.DataFrame:
    """Analyze strategy performance."""
    return pd.DataFrame({
        'strategy': str,
        'success_rate': float,
        'avg_utility': float,
        'avg_turns': float,
        'adherence_score': float
    })
```

5. Result Storage

5.1 Checkpoint Format

```
{
    'state': {
        'experiment_name': str,
        'config': dict,
        'progress': dict
    },
    'completed_negotiations': {
        'scenario_id': {
            'metrics': dict,
            'history': list
        }
    },
    'model_pair_metrics': {
        'model_pair_key': {
            'statistics': dict,
            'analysis': dict
        }
    }
}
```

5.2 Final Results Format

```
{
    'experiment_name': str,
    'config': dict,
    'summary': {
        'total_scenarios': int,
        'completed': int,
        'failed': int,
        'duration': float
    },
    'model_pair_metrics': dict,
    'completed_negotiations': dict,
    'strategy_analysis': dict,
    'error_analysis': dict
}
```

6. Performance Metrics

6.1 Success Metrics

```
def compute_success_metrics(results: Dict) -> Dict:
    """Compute core success metrics."""
    return {
        'deal_rate': float,  # % of successful deals
        'avg_utility': float,  # average combined utility
        'efficiency': float,  # turns to completion
```

```
        'fairness': float    # deal balance score
    }
```

6.2 Time Metrics

```
def compute_time_metrics(results: Dict) -> Dict:
    """Compute timing metrics."""
    return {
        'avg_turn_time': float,
        'avg_negotiation_time': float,
        'timeouts': int,
        'response_distribution': Dict[str, float]
    }
```

7. Recovery & Fault Tolerance

7.1 Checkpoint Management

```
class CheckpointManager:
    """Manages experiment checkpoints."""

    def save_checkpoint(self):
        """Save current state."""
        pass

    def load_checkpoint(self, path: str):
        """Restore from checkpoint."""
        pass

    def list_checkpoints(self) -> List[str]:
        """List available checkpoints."""
        pass
```

7.2 Error Recovery

```
def handle_failure(
    error: Exception,
    context: Dict
) -> None:
    """Handle experiment failure."""
    # Log error
    logger.error(f"Experiment failed: {str(error)}")

    # Save state
    save_emergency_checkpoint()

    # Notify if configured
    send_notification(error, context)
```

8. Visualization Preparation

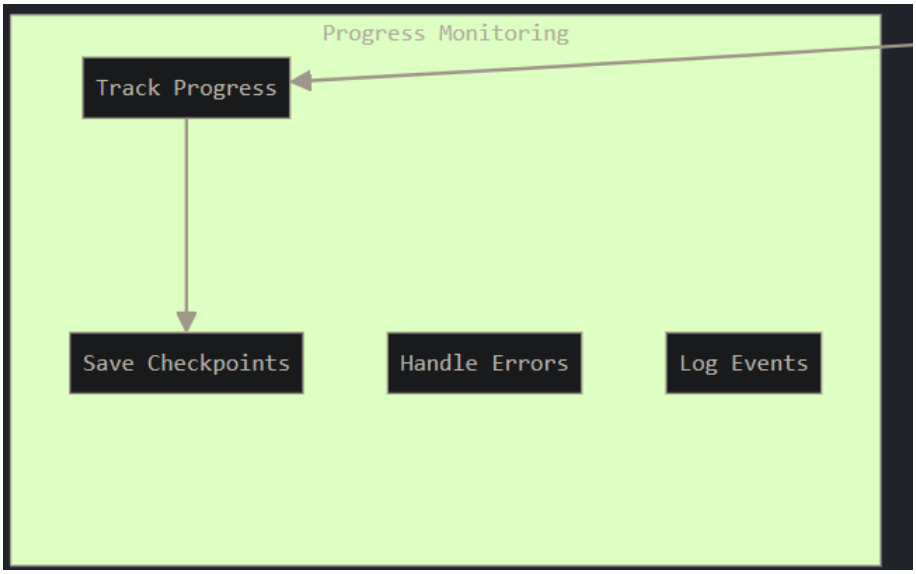
8.1 Data Transformation

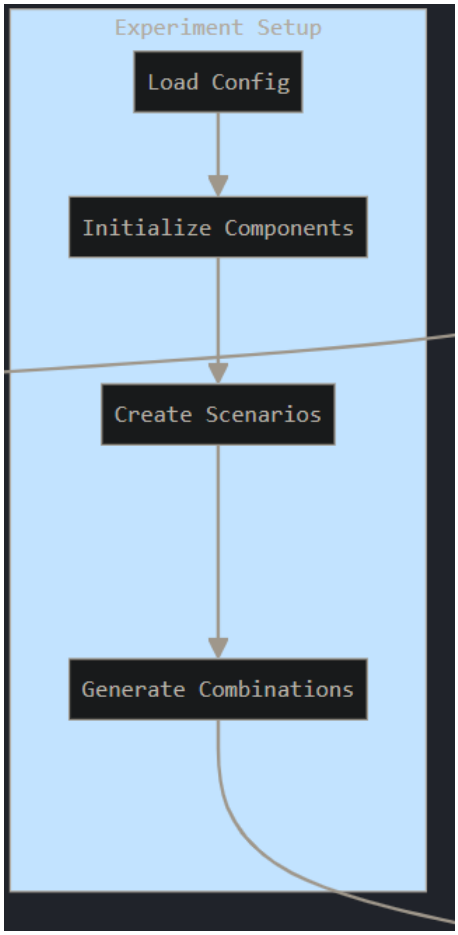
```
def prepare_visualization_data(
    results: Dict
) -> Dict[str, pd.DataFrame]:
    """Prepare data for visualization."""
    return {
        'outcomes': _prepare_outcome_data(),
        'strategies': _prepare_strategy_data(),
        'models': _prepare_model_data(),
        'timing': _prepare_timing_data()
    }
```

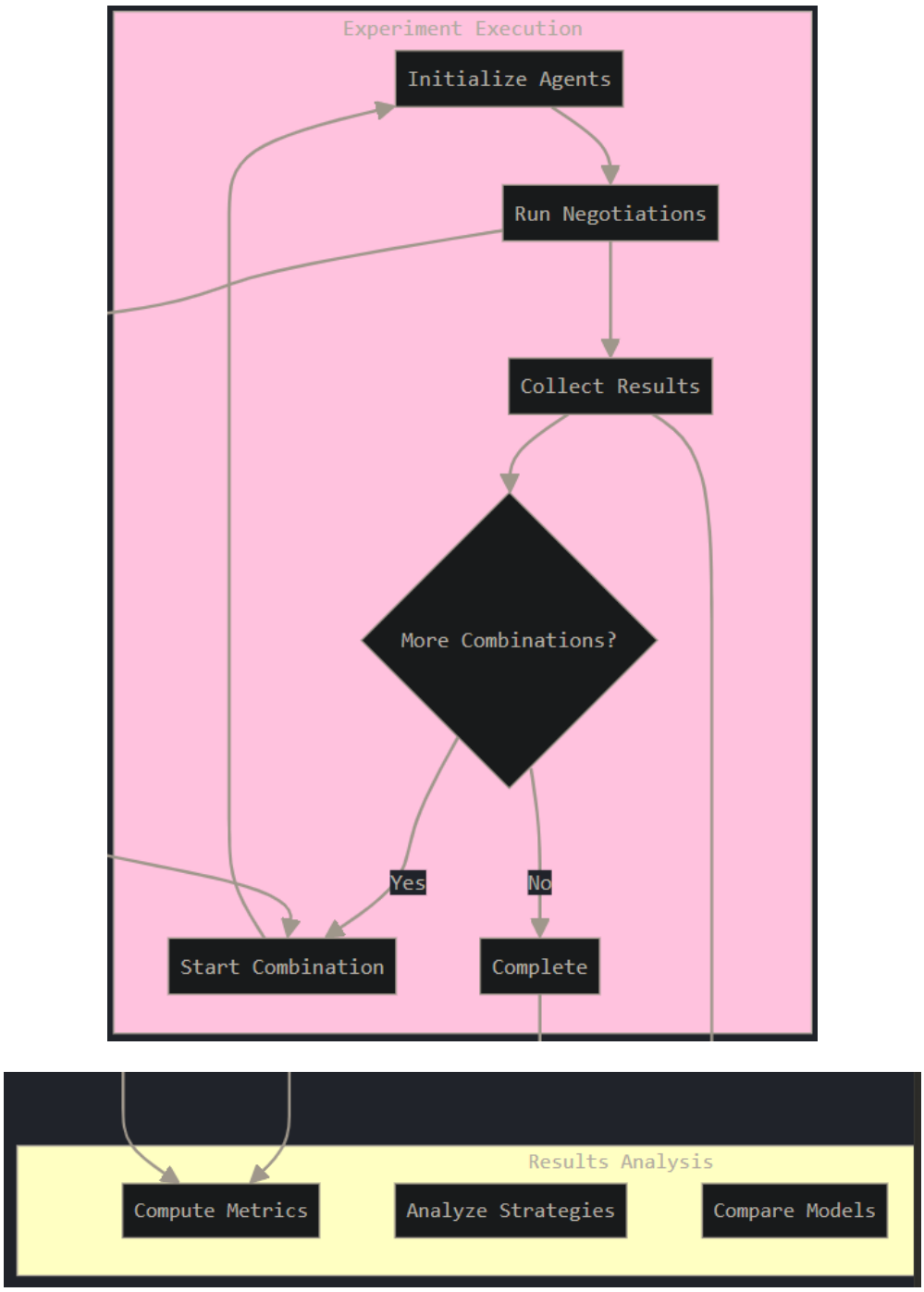
8.2 Export Formats

```
def export_results(
    results: Dict,
    format: str = 'all'
) -> None:
    """Export results in specified format."""
    formats = {
        'csv': _export_csv,
        'json': _export_json,
        'excel': _export_excel
    }

    if format == 'all':
        for export_fn in formats.values():
            export_fn(results)
    else:
        formats[format](results)
```


Progress Monitoring
Track Progress
Save Checkpoints
Handle Errors
Log Events


Experiment Setup
Load Config
Initialize Components
Create Scenarios
Generate Combinations


Experiment Execution
Initialize Agents
Run Negotiations
Collect Results
More Combinations?
Yes
No
Start Combination
Complete
Results Analysis
Compute Metrics
Analyze Strategies
Compare Models