1. Introduction

The emergence of Large Language Models (LLMs) as intelligent runtime environments has fundamentally altered the landscape of software architecture. Where traditional software systems require precise, imperative code to specify every execution step, LLM-based systems can interpret natural language descriptions of intent and autonomously determine how to achieve specified goals. This capability enables a new class of software: Declarative Multi-Agent Systems (MAS), where multiple AI agents collaborate to accomplish complex tasks through natural language blueprints rather than compiled code.

However, this nascent field lacks a shared vocabulary for design. Development teams building declarative MAS repeatedly encounter similar challenges—how to define agent behavior, how agents should communicate, how to ensure output quality, how to prevent AI hallucinations—yet each team develops ad hoc solutions without the benefit of accumulated collective wisdom. This situation mirrors the state of object-oriented programming before the publication of design patterns: practitioners solving the same problems independently, unable to efficiently transfer hard-won insights.

This paper presents POMASA (Pattern-Oriented Multi-Agent System Architecture), a pattern language for constructing declarative multi-agent systems. POMASA is neither a framework (it imposes no mandatory code structure) nor a tool (the generator we provide is merely one way to apply it). At its core, POMASA is a pattern language—a network of interconnected design patterns that guide both humans and AI in building MAS.

The pattern language comprises 15 patterns organized into four categories: Core patterns (COR) that define fundamental system characteristics, Structure patterns (STR) that organize static system architecture, Behavior patterns (BHV) that define dynamic system behavior, and Quality patterns (QUA) that ensure system reliability. Each pattern is further classified by necessity: must patterns that every system requires, recommended patterns that most systems benefit from, and optional patterns for specific scenarios.

Beyond documenting patterns, this paper advances a key insight about the nature of pattern languages in the AI era: **for AI runtime environments, pattern languages themselves become executable**. Unlike traditional pattern languages that serve as design guidance requiring human translation into code, POMASA patterns can be directly interpreted by AI systems to generate working MAS implementations. This property transforms the meaning of "open source"—publishing a pattern language paper becomes equivalent to releasing source code, as readers can present the patterns to an AI system and reconstruct the architecture.

We demonstrate POMASA through an industry analysis system that employs 8 agents to conduct systematic research on industrial sectors. This system processes information through a pipeline of collection, verification, analysis, and synthesis, ultimately producing academic-quality reports with full data lineage. The system has been deployed in practice, producing substantive analysis reports that demonstrate the viability of the pattern language approach.

The remainder of this paper proceeds as follows. Section 2 introduces the industry analysis system that serves as our running example. Section 3 describes the pattern language's organizational structure. Section 4 presents eight essential patterns in detail. Section 5 demonstrates how patterns combine to form a working system. Section 6 elaborates the key insight about executable pattern languages. Section 7 discusses implications, limitations, and concludes.

2. Background: The Industry Analysis System

To ground our discussion of patterns in concrete reality, we introduce an industry analysis system built using POMASA. This system conducts systematic research on industrial sectors, analyzing them through a theoretical framework that examines ownership structures, distribution mechanisms, resource allocation, and economic governance. The system has produced substantive reports, including a comprehensive analysis of the electric vertical takeoff and landing (eVTOL) industry that spans multiple dimensions of institutional analysis.

2.1 System Overview

The industry analysis system transforms a research question into an academic-quality report through a multi-stage pipeline. Given an industry identifier and a theoretical framework, the system:

1. **Scans** the industry landscape to establish basic understanding and generate targeted research questions
2. **Researches** each question through systematic web search, collecting policies, statistics, cases, and expert views
3. **Verifies** collected data in an independent context to identify and eliminate AI hallucinations
4. **Analyzes** each research dimension against the theoretical framework
5. **Synthesizes** individual analyses into coherent higher-level insights
6. **Generates** a final report with executive summary
7. **Reviews** the complete output against quality standards

2.2 System Scale and Complexity

The system comprises 8 distinct agents, each defined by a natural language blueprint averaging 7-9 KB of structured markdown. The theoretical framework encompasses 4 dimensions, 12 features, and 55 functional items, each requiring independent research and analysis. A complete run produces:

- Initial industry overview documents covering basic profile, ownership structure, policy environment, and development history

- Research questions organized by dimension (approximately 200 specific questions across 55 functional items)
- Deep research materials including policies, statistics, cases, and expert views for each functional item
- Verification reports identifying unreliable data
- Analysis documents for each functional item, feature, and dimension
- Final report with executive summary (the eVTOL report spans approximately 50,000 words)

2.3 Why This Case Study

We selected this system as our running example for several reasons:

Complexity: The system involves genuine multi-agent coordination, not merely sequential prompts. Agents must pass data through file-based interfaces, maintain data lineage across stages, and coordinate parallel execution of 55 independent research tasks.

Real Deployment: The system has been used in practice, producing reports that have undergone human review. We can point to concrete outputs rather than hypothetical scenarios.

Pattern Coverage: Building this system required solving most challenges that declarative MAS developers face: agent definition, inter-agent communication, quality assurance, hallucination prevention, parallel execution, and orchestration.

Verifiable Quality: The eVTOL analysis report demonstrates that the pattern language approach can produce substantive, academically-rigorous output. The report includes 150+ citations to verifiable sources, structured analysis across multiple dimensions, and actionable policy recommendations—quality markers that validate the underlying architecture.

2.4 A Glimpse of the Output

To illustrate what this system produces, consider this excerpt from the executive summary of the eVTOL industry analysis:

> The eVTOL industry exhibits clear functional division and effective synergy among three ownership forms: state-owned economy plays a "stabilizing anchor" role, maintaining comprehensive control over airspace resources, airworthiness certification, infrastructure construction, and industrial fund investment (exceeding 100 billion yuan); public land ownership enables "planning leadership," with rapid deployment of 1,200 takeoff/landing points in Shenzhen, 400 routes in Shanghai, and the world's first cross-sea cross-city route—achievements difficult to replicate in countries with private land ownership; non-public economy serves as "innovation engine,"

> with private enterprises accounting for 68.1%-81% of aircraft manufacturers, 40 companies simultaneously conducting R&D with diversified technical routes, and patent applications accounting for 58.3% globally.

This excerpt demonstrates several pattern outcomes: data grounded in verifiable sources (the percentages and counts), multi-dimensional analysis (ownership structure as one of four dimensions), and synthesis that goes beyond mere data compilation to identify institutional patterns. The full report maintains this quality across 55 functional items and four dimensions, enabled by the systematic application of POMASA patterns.

3. The Pattern Language Structure

POMASA organizes its patterns through two orthogonal classification schemes: a categorical taxonomy that groups patterns by their architectural concern, and a necessity hierarchy that indicates how critical each pattern is to system viability.

3.1 Categorical Taxonomy

Patterns are classified into four categories, each identified by a three-letter prefix:

COR (Core): Patterns that define the fundamental characteristics of declarative MAS. These patterns establish what makes a system "declarative" and "multi-agent" in the AI era. Without Core patterns, the system would not qualify as a declarative MAS. Currently: 2 patterns.

STR (Structure): Patterns that organize the static architecture of the system—how components are arranged, how data is stored, how boundaries are defined. Structure patterns determine the "shape" of the system at rest. Currently: 6 patterns.

BHV (Behavior): Patterns that govern dynamic system behavior—how agents execute, how they coordinate, how data flows through the pipeline. Behavior patterns determine what happens when the system runs. Currently: 4 patterns.

QUA (Quality): Patterns that ensure system reliability and output quality—how to embed standards, how to verify data, how to prevent hallucinations. Quality patterns determine whether the system produces trustworthy results. Currently: 3 patterns.

3.2 Necessity Hierarchy

Orthogonal to categorical classification, each pattern carries a necessity level:

Must: Patterns without which the system cannot function correctly. Every declarative MAS must implement these patterns. Currently: 4 patterns (COR-01, COR-02, BHV-02, QUA-03).

Recommended: Patterns that most systems benefit from significantly. Omitting these patterns is possible but usually results in a weaker system. Currently: 8 patterns.

Optional: Patterns that address specific scenarios or provide additional capabilities. Systems should adopt these based on their particular requirements. Currently: 3 patterns.

3.3 The Pattern Matrix

Combining these two dimensions yields the following matrix:

Category	Must	Recommended	Optional
COR	COR-01: Prompt-Defined Agent COR-02: Intelligent Runtime	—	—
STR	—	STR-01: Reference Data Configuration STR-02: Filesystem Data Bus STR-03: Workspace Isolation STR-04: Business-Driven Agent Design STR-05: Composable Document Assembly STR-06: Methodological Guidance	—
BHV	BHV-02: Faithful Agent Instantiation	BHV-01: Orchestrated Agent Pipeline	BHV-03: Parallel Instance Execution BHV-04: Progressive Data Refinement

Category	Must	Recommended	Optional
QUA	QUA-03: Verifiable Data Lineage	QUA-01: Embedded Quality Standards	QUA-02: Layered Quality Assurance

3.4 Pattern Relationships

Patterns do not exist in isolation; they form a network of dependencies and reinforcements:

```
         COR-02
       Intelligent
         Runtime
             |
             | enables
             v
         COR-01
    Prompt-Defined Agent
        /    |    \
       /     |     \
      v      v      v
 STR-01   BHV-01   QUA-01
Reference Pipeline Quality
   Data     |    Standards
     |      |         |
     v      v         v
 STR-06   BHV-02   QUA-03
Method.  Faithful  Data
Guidance  Inst.   Lineage
            |
            v
         STR-02
       Filesystem
        Data Bus
            |
            v
         STR-03
        Workspace
        Isolation
```

The diagram reveals the architecture's foundation: COR-02 (Intelligent Runtime) enables COR-01 (Prompt-Defined Agent), which in turn makes all other patterns possible. The three branches—reference data management, execution coordination, and quality assurance—represent the primary architectural concerns that Structure, Behavior, and Quality patterns address.

3.5 Reading the Patterns

Each pattern in the following section follows a consistent format derived from the pattern language tradition:

- **Context**: The situation in which the pattern applies
- **Problem**: The design challenge being addressed
- **Forces**: The competing concerns that make the problem difficult
- **Solution**: The recommended approach
- **Example**: Concrete illustration from the industry analysis system
- **Discussion**: Consequences, limitations, and related considerations

This format makes patterns comparable and composable—readers can evaluate whether a pattern's context matches their situation and weigh the forces against their priorities.

4. Essential Patterns

This section presents eight patterns essential to building declarative multi-agent systems, organized by the logical sequence of system construction: runtime foundation → agent definition → knowledge organization → coordination mechanisms → data flow → quality assurance.

4.1 COR-02: Intelligent Runtime

Context: You are building a system where agents are defined declaratively through natural language descriptions rather than imperative code.

Problem: Who executes the declaratively-defined agent behavior?

Traditional runtime environments (JVM, Python interpreter, shell) mechanically execute code instructions. They cannot understand natural language descriptions of intent. A declarative agent blueprint requires an execution environment capable of semantic understanding, decision-making, and adaptive behavior.

Forces:

- *Intelligence vs. Controllability*: The more intelligent the runtime, the less predictable its behavior
- *Adaptability vs. Consistency*: Ability to adapt to changes may cause execution inconsistency
- *Autonomy vs. Auditability*: Autonomous decision-making makes auditing difficult
- *Capability vs. Cost*: Intelligent runtimes require AI model access, incurring costs

Solution: Use an AI system with comprehension and decision-making capabilities as the runtime environment. The intelligent runtime understands natural language blueprints, selects execution methods, evaluates output quality, and handles exceptional situations.

The intelligent runtime operates at four capability layers:

1. **Basic Execution**: File system operations, network access, tool invocation
2. **Execution Decisions**: Tool selection and composition, parameter inference, error handling
3. **Quality Evaluation**: Understanding quality standards, assessing output quality, self-correction
4. **Meta-cognition**: Progress tracking, strategy optimization, anomaly identification

The fundamental distinction from traditional runtimes:

Dimension	Traditional Runtime	Intelligent Runtime
Input	Bytecode/source code	Natural language blueprint
Execution	Mechanical instruction sequence	Intent understanding, intelligent method selection
Decision-making	None (determined by code)	Strong (adaptive decisions at runtime)
Error handling	Throw exception, halt	Understand error context, attempt recovery
Determinism	High (same input $\rightarrow$ same output)	Medium (same blueprint may take different paths)

Example: In the industry analysis system, Claude Code serves as the intelligent runtime. When the Initial Scanner blueprint states "collect basic industry information including scale, market structure, ownership composition, and policy environment," the runtime:

1. Decides to use web search for each topic
2. Selects appropriate search queries based on the industry
3. Evaluates whether retrieved information is sufficient
4. Adapts strategy if certain information proves difficult to find
5. Formats output according to the blueprint's specifications

None of these decisions are explicitly programmed—the runtime interprets intent and acts accordingly.

Discussion: The intelligent runtime pattern introduces platform dependency. Currently, Claude Code provides the most mature implementation, but this creates lock-in. Blueprint designers should avoid relying on runtime-specific features when possible, maintaining portability for future runtime alternatives.

The non-determinism of intelligent runtimes challenges traditional software engineering assumptions. The same blueprint may execute differently across runs. This trade-off—flexibility for determinism—defines the declarative MAS paradigm.

4.2 COR-01: Prompt-Defined Agent

Context: You have an intelligent runtime (COR-02) capable of understanding and executing natural language instructions.

Problem: How do you define AI agent behavior?

Traditional approaches use code to specify every logical step. But when task complexity exceeds what fixed logic can express—when data sources have varying structures, external environments change unpredictably, and "quality" is a semantic concept—hardcoded rules become brittle. Quality standards like "objective, professional, conforming to academic norms" are semantic-level requirements that resist expression in procedural code.

Forces:

- *Flexibility vs. Predictability*: Declarative definitions enable flexibility but reduce execution certainty
- *Expressiveness vs. Ambiguity*: Natural language is expressive but may introduce ambiguity
- *Development efficiency vs. Runtime cost*: Development is highly efficient, but each execution consumes AI resources
- *Human readability vs. Machine executability*: Must find expression that serves both

Solution: Use natural language documents (Agent Blueprints) to describe what the agent "should do," "what standards to follow," and "what outcomes to achieve"—rather than specifying "exactly how to do it."

A blueprint's essential structure:

```
# Agent Name

## Workspace Isolation Requirements
[Workspace constraints per STR-03]

## Your Role
[One paragraph defining identity and core responsibility]

## Prerequisites
[Reference materials to read before execution]

## Input Parameters
- `{PARAM_1}`: [Description]
- `{PARAM_2}`: [Description]
```

```
## Workflow

### Phase One: [Phase Name]
[Goals, strategies, considerations for this phase]

### Phase Two: [Phase Name]
...

## Output Requirements

**Output Location**: `path/to/output/`

**Output Format**:
[Detailed file structure and format specification]

## Completion Criteria
- [ ] [Criterion 1]
- [ ] [Criterion 2]
- ...

## Notes
[Special reminders, edge cases, prohibitions]
```

Key design principles:

1. **Declare intent, not details**: Describe desired outcomes rather than specific steps
2. **Support parameterization**: Use placeholders (e.g., `{INDUSTRY_ID}`) for instantiation
3. **Embed quality standards**: Write quality requirements into the blueprint
4. **Maintain readability**: Use structured Markdown that non-programmers can understand

Example: The Deep Researcher agent in the industry analysis system begins:

```
# Deep Researcher

## Your Role

You are the deep research specialist for the ESSCC industry analysis
project. Your responsibility is to conduct comprehensive, in-depth
research on a specific functional item, collecting policies, statistics,
cases, and expert views that provide evidence for subsequent analysis.

## Input Parameters

- `{INDUSTRY_ID}`: Industry identifier
- `{FUNCTION_ID}`: Functional item identifier (e.g., "1.1")
- `{FUNCTION_NAME}`: Functional item name

## Workflow

### Research Strategy

For each functional item, collect four types of information:
1. **Policies**: Government policies, regulations, plans related to this function
2. **Statistics**: Quantitative data demonstrating this function
3. **Cases**: Concrete examples illustrating this function in practice
4. **Expert Views**: Academic or industry expert opinions on this function

### Quality Requirements

- Every piece of information must have a verifiable source (URL)
- Prioritize authoritative sources (government, academic, major industry)
- Cross-verify important claims across multiple sources
```

This blueprint tells the agent *what* to collect and *what standards* to meet, not *how* to search or *which websites* to visit.

Discussion: Prompt-defined agents trade determinism for adaptability. The same blueprint may produce different execution paths—a feature, not a bug. The agent adapts to what it finds rather than failing when reality doesn't match expectations.

The pattern also dramatically lowers the barrier to system maintenance. Domain experts who cannot program can read, understand, and modify agent blueprints. This democratizes system evolution.

4.3 STR-01: Reference Data Configuration

Context: You have agents defined through blueprints (COR-01) that need external knowledge to perform their tasks.

Problem: How do you provide agents with the external knowledge they need to execute tasks?

AI agents require two types of external knowledge:

1. **Domain knowledge**: What things are—theoretical frameworks, concept definitions, classification systems
2. **Methodological guidance**: How to do things—data sources, analysis methods, output formats

Embedding this knowledge directly in agent blueprints creates bloated blueprints, scattered knowledge, difficult updates, and excludes domain experts from maintenance.

Forces:

- *Centralization vs. Distribution*: Centralized management aids maintenance but may create single points of failure
- *Structure vs. Flexibility*: Structured formats enable machine processing but limit expressive freedom
- *Completeness vs. Conciseness*: Complete knowledge bases are large; concise ones may omit important details
- *Generality vs. Specificity*: General knowledge applies broadly; specific knowledge is more precise

Solution: Externalize knowledge as independent data files, separate from agent blueprints. Blueprints reference these files; agents read and apply the knowledge at execution time.

Organize reference data into two subdirectories:

```
references/
+-- domain/                    # Domain knowledge: what things are
|   +-- theoretical_framework.md    # Theoretical framework
|   +-- concepts.md                 # Concept definitions
|   `-- literature_review.md        # Background literature
|
`-- methodology/               # Methodological guidance: how to do things
    +-- research-overview.md        # Research objectives and scope
    +-- data-sources.md             # Data source guide
    +-- analysis-methods.md         # Analysis methods
    `-- output-template.md          # Output format template
```

Critical principle: User-provided domain reference materials must be preserved in full—never summarized or condensed. AI systems tend to "helpfully" summarize lengthy documents, but this:

- Loses information (summaries inevitably omit details that may prove crucial)
- Introduces bias (AI understanding may be skewed)
- Destroys authority (original materials carry expert authority that summaries lack)
- Breaks traceability (summaries prevent tracing back to original statements)

Example: The industry analysis system's reference data includes:

```
references/
+-- domain/
|   +-- theoretical_framework_explained.md   # 15KB framework explanation
|   `-- theoretical_framework_literature_review.md  # Academic grounding
|
`-- methodology/
    +-- research-overview.md    # What questions to answer
    +-- data-sources.md         # Where to find data, credibility tiers
    +-- analysis-methods.md     # Five core questions per functional item
    `-- output-template.md      # Report structure and formatting
```

The `theoretical_framework_explained.md` file contains the complete ESSCC framework—4 dimensions, 12 features, 55 functional items—each with definitions, manifestations, and evaluation criteria. This is the domain knowledge that every agent consults.

Discussion: Reference data configuration enables separation of concerns. Domain experts maintain domain knowledge; methodology experts maintain methodological guidance; system developers maintain agent blueprints. Each can evolve independently.

The pattern also supports system reuse. The same agent blueprints can be applied to different domains by swapping reference data—a different theoretical framework yields a different analysis system.

4.4 BHV-01: Orchestrated Agent Pipeline

Context: You have multiple agents defined through blueprints (COR-01) that need to collaborate to accomplish a complex task.

Problem: How do multiple agents coordinate to accomplish tasks that exceed any single agent's scope?

Complex tasks like "analyze an industry" cannot be accomplished by a single agent in a single execution. The task naturally decomposes into stages: first scan the landscape, then research details, then analyze findings, then synthesize conclusions, then generate reports. Each stage has different requirements and produces different outputs.

Forces:

- *Autonomy vs. Coordination*: Agents need freedom to execute but must align toward common goals
- *Parallelism vs. Dependencies*: Some work can proceed in parallel; other work depends on prior results
- *Simplicity vs. Capability*: Simple coordination is easier to implement but limits what the system can achieve
- *Centralization vs. Distribution*: Central orchestration enables control but creates bottlenecks

Solution: Decompose tasks into sequential stages, each handled by specialized agent types, coordinated by an Orchestrator agent.

The pipeline structure:

```
Orchestrator
    +-- Stage 1: Data Collection (parallel)
    |   +-- Agent Instance 1 → output/stage1/item1/
    |   +-- Agent Instance 2 → output/stage1/item2/
    |   `-- Agent Instance N → output/stage1/itemN/
    |
    +-- Stage 2: Analysis (parallel)
    |   `-- Agent Instance M → output/stage2/
    |
    `-- Stage 3: Report Generation (sequential)
        `-- Agent Instance 1 → output/stage3/
```

The Orchestrator agent:

- Reads the master plan and reference data
- Initializes the data directory structure
- Launches each stage in sequence
- Passes parameters to stage agents
- Validates stage completion before proceeding
- Handles exceptions and coordinates retries

Example: The industry analysis system's orchestrator (`00.orchestrator.md`) coordinates seven stages:

```
## Execution Flow

### Stage 0: Initialization
- Verify reference files exist and are complete
- Create data directory structure for this industry

### Stage 1: Initial Scanning
- Launch Initial Scanner agent
- Parameters: INDUSTRY_ID, INDUSTRY_NAME
- Wait for completion; verify outputs exist

### Stage 2: Deep Research
- For each of 55 functional items:
  - Launch Deep Researcher agent
  - Parameters: INDUSTRY_ID, FUNCTION_ID, FUNCTION_NAME
- Can execute in parallel
- Wait for all to complete

### Stage 3: Data Verification
- Launch Data Verifier agent in independent context
- Verify all collected data; eliminate hallucinations
- Wait for completion; note which data was eliminated

### Stage 4: Analysis
- For each functional item:
  - Launch Analyzer agent
  - Parameters: INDUSTRY_ID, FUNCTION_ID
- Can execute in parallel
- Wait for all to complete

### Stage 5: Synthesis
- Launch Synthesizer agent
- Aggregate functional item analyses into feature and dimension syntheses
- Wait for completion

### Stage 6: Report Generation
- Launch Reporter agent
- Generate final report and executive summary
- Wait for completion

### Stage 7: Quality Review
- Launch Quality Checker agent
- Final quality verification
- Wait for completion
```

Discussion: The orchestrated pipeline pattern trades flexibility for predictability. The fixed stage sequence constrains what the system can do but makes behavior comprehensible and debuggable. When something goes wrong, you know which stage failed.

The pattern also enables incremental execution. If Stage 3 fails, Stages 1 and 2 outputs remain intact. The system can resume from the failed stage rather than starting over.

4.5 BHV-02: Faithful Agent Instantiation

Context: You have an Orchestrator (BHV-01) that needs to invoke other agents to execute portions of the task.

Problem: How do you ensure agents are correctly instantiated and executed?

In multi-agent systems, orchestrators invoke other agents to execute tasks. Common mistakes include:

1. **Summary relay**: The orchestrator reads Agent B's blueprint, then summarizes it to launch a subagent. Critical details are lost; execution deviates.
2. **Task bundling**: When executing batch tasks, the orchestrator bundles multiple independent tasks into a single subagent call. Each task receives inadequate execution quality.

These shortcuts may seem harmless in short call chains, but in long chains or large batches, they cause severe quality degradation.

Forces:

- *Efficiency vs. Quality*: Summary relay or task bundling may seem "efficient" but degrades quality
- *Brevity vs. Completeness*: Brief invocations may omit critical information
- *Flexibility vs. Rigor*: Flexible invocation patterns are harder to keep consistent

Solution: Every agent instance must directly read and execute the complete blueprint. The caller passes only parameters, never blueprint content. Each independent task corresponds to an independent subagent invocation. The orchestrator must verify results against blueprint completion criteria.

Core principles:

1. **Caller passes parameters only, never relays blueprint**: The orchestrator tells the subagent which blueprint file to read. The subagent reads the complete blueprint and executes. No summarizing, simplifying, or relaying.
2. **One task instance = one subagent invocation**: In batch tasks, each independent task launches a separate subagent. Do not bundle multiple tasks into one invocation.
3. **Blueprint is the sole basis for execution**: Subagent behavior is determined entirely by the blueprint. No dependence on caller's "interpretation" or "guidance."
4. **Orchestrator must validate results**: When receiving subagent results, verify against blueprint completion criteria item by item. Do not accept results that fail criteria; require redo or report exception.
5. **Completion criteria are non-negotiable**: Blueprint completion criteria are hard requirements. Subagents must not unilaterally lower standards. When facing difficulties, they must consult the orchestrator rather than cutting corners.

Example: The orchestrator's invocation of Deep Researcher:

```
## Stage 2: Deep Research

For each functional item in the question list:

1. Launch a new subagent
2. Instruct the subagent:
   - Read `agents/02.deep_researcher.md`
   - Execute strictly according to the blueprint
   - Parameters: INDUSTRY_ID={current industry}, FUNCTION_ID={current item}
3. Wait for subagent completion
4. Verify outputs against completion criteria

**Important**:
```

```
- Each functional item launches an independent subagent
- Do NOT bundle multiple items into one subagent
- Do NOT summarize or relay blueprint content
```

Standard invocation wording:

```
Please launch a subagent to perform the following task:
1. Read agents/02.deep_researcher.md
2. Execute strictly according to that blueprint
3. Parameters:
   - INDUSTRY_ID: evtol
   - FUNCTION_ID: 1.1
   - FUNCTION_NAME: Control Strategic Sectors

**Important constraints**:
- Completion criteria in the blueprint are hard requirements
- Do not lower standards for any reason
- If difficulties arise, report immediately rather than adopting workarounds
```

Discussion: This pattern emerged from debugging production failures. Early versions of the industry analysis system suffered from quality degradation in long call chains—by the time instructions passed through multiple orchestration levels, critical requirements had been lost through successive summarization.

The principle that completion criteria are non-negotiable was added after observing agents "helpfully" reducing scope when facing difficulties. An agent tasked with researching 55 items might decide to sample 10 "representative" items—a reasonable-seeming adaptation that destroys the analysis's comprehensiveness.

4.6 STR-02: Filesystem Data Bus

Context: You have multiple agents (COR-01) coordinated through a pipeline (BHV-01) that need to exchange data.

Problem: How do agents pass data to each other?

Multi-agent systems require data exchange between agents. Traditional solutions (API calls, message queues, shared databases) require additional infrastructure, complex configuration, and network programming. AI multi-agent systems need a data passing mechanism that naturally fits the runtime, is fully traceable, is human-readable and editable, and requires minimal deployment complexity.

Forces:

- *Simplicity vs. Functionality*: File systems are simple but lack transactions, queries, and other advanced features
- *Transparency vs. Performance*: All data being visible enables transparency but increases I/O overhead

- *Loose coupling vs. Real-time*: File-based loose coupling cannot support real-time communication
- *Human readability vs. Machine efficiency*: Text formats are human-friendly but less efficient to process

Solution: Use the filesystem as the data passing medium. Agents read and write JSON and Markdown files through agreed-upon file paths. They do not communicate directly.

The directory structure reflects data flow:

```
data/
+-- {INSTANCE_ID}/              # Partition by run instance
|   +-- 01.materials/           # Stage 1 output: raw data
|   |   +-- {entity_1}/         # Partition by processing entity
|   |   +-- {entity_2}/
|   |   `-- ...
|   |
|   +-- 02.analysis/            # Stage 2 output: analysis results
|   |   `-- ...
|   |
|   `-- 03.reports/             # Stage 3 output: final reports
|       `-- ...
```

Core design principles:

1. **Directory structure is data flow**: Directory hierarchy reflects processing stages; subdirectories reflect data partitions

2. **File format conventions**: Structured data uses JSON; textual content uses Markdown; metadata uses YAML front matter

3. **Path conventions**: Use consistent naming conventions; paths can be derived from parameters without explicit configuration

4. **Data immutability**: Each stage produces new data without modifying prior stages; preserve complete data evolution history

Example: The industry analysis system's data directory for eVTOL:

```
data/evtol/
+-- 01.materials/
|   +-- 01.industry_overview/
|   |   +-- basic_profile.md
|   |   +-- ownership_structure.md
|   |   +-- policy_environment.md
|   |   `-- source_list.md
|   +-- 02.question_list/
|   |   +-- dimension_1_ownership.md
|   |   `-- ...
|   `-- 03.deep_research/
|       +-- 1.1_control_strategic_sectors/
|       |   +-- policies.md
|       |   +-- statistics.md
|       |   +-- cases.md
|       |   `-- source_list.md
|       `-- ...
|
+-- 02.analysis/
|   +-- functions/
|   +-- features/
|   `-- dimensions/
|
`-- 03.reports/
    +-- final_report.md
    `-- executive_summary.md
```

Data flows through the structure:

- Initial Scanner → `01.materials/01.industry_overview/`, `01.materials/02.question_list/`
- Deep Researcher → `01.materials/03.deep_research/`
- Analyzer → `02.analysis/functions/`, `02.analysis/features/`, `02.analysis/dimensions/`
- Reporter → `03.reports/`

Discussion: The filesystem data bus pattern optimizes for debuggability over performance. Every intermediate artifact is visible, inspectable, and editable. When analysis produces unexpected results, developers can examine exactly what data each agent received and produced.

The pattern also enables human intervention. A domain expert can review and correct deep research outputs before analysis proceeds. The system accommodates human-in-the-loop workflows naturally.

The main limitation is performance. File I/O is slower than in-memory communication. For systems requiring high throughput or real-time responses, this pattern is inappropriate. The industry analysis system, where a complete run takes hours and throughput is measured in reports per day, finds this trade-off acceptable.

4.7 STR-03: Workspace Isolation

Context: You have agents (COR-01) that read and write files through the filesystem data bus (STR-02).

Problem: How do you prevent agents from accessing or modifying files they shouldn't?

AI agents have file system read/write capabilities. Without restrictions, they may:

- Accidentally read files from other projects, contaminating context
- Accidentally modify system files or other projects' data
- Cause interference between different projects
- Make system behavior depend on state outside the project

Forces:

- *Security vs. Convenience*: Access restrictions increase security but may add inconvenience
- *Isolation vs. Sharing*: Complete isolation may prevent legitimate resource sharing
- *Explicit constraints vs. Implicit assumptions*: Explicit constraints are more reliable but add configuration burden

Solution: At the beginning of every agent blueprint, explicitly declare workspace boundaries. Forbid reading or writing any files outside the specified directory.

Standard constraint declaration:

```
## Workspace Isolation Requirements

**IMPORTANT**: You must work ONLY within the project directory `{PROJECT_PATH}/`.
- You are **forbidden** from reading any files outside this directory
- You are **forbidden** from writing any files outside this directory
- All file paths you use must be relative to this project root or absolute paths within this
- This constraint ensures system isolation and prevents context contamination
```

Placement: The constraint declaration should appear at the very beginning of the blueprint (immediately after the title), ensuring it is:

1. Read first by the agent
2. Emphasized in importance
3. Established before any concrete task

Example: Every agent in the industry analysis system begins with:

```
# Initial Scanner

## Workspace Isolation Requirements

**IMPORTANT**: You must work ONLY within the project directory `industry_assessment/`.
- You are **forbidden** from reading any files outside this directory
- You are **forbidden** from writing any files outside this directory
- All file paths you use must be relative to this project root
- This constraint ensures system isolation and prevents context contamination

---
```

```
## Your Role
...
```

Discussion: Workspace isolation relies on convention rather than technical enforcement. The intelligent runtime (Claude Code) respects these constraints but could technically violate them. The pattern works because the AI system genuinely attempts to follow instructions, not because violations are technically impossible.

This convention-based approach has an important benefit: the constraint is visible and auditable. Reviewers can see that the blueprint includes isolation requirements. Technical enforcement mechanisms, by contrast, operate invisibly—you must trust that they work correctly.

4.8 QUA-03: Verifiable Data Lineage

Context: You have a multi-agent system that collects, processes, and synthesizes data to produce analytical outputs.

Problem: How do you ensure that data and conclusions produced by the AI system are trustworthy?

AI systems, especially LLMs, suffer from severe "hallucination" problems: they may fabricate non-existent data, invent URLs, and create fictitious citations. Even when blueprints explicitly require data verification, within the same execution context, AI often cannot effectively identify hallucinations it produced earlier.

This problem is especially severe in research-oriented MAS because:

- Final outputs require academic-level credibility
- Erroneous data leads to erroneous conclusions
- Hallucinated data, once entering the analysis stage, gets "laundered" into seemingly reasonable arguments

Forces:

- *Efficiency vs. Rigor*: Strict verification increases time cost
- *Trust vs. Verification*: Cannot blindly trust AI output; need independent verification
- *Completeness vs. Credibility*: Better to have less data that is credible than more data that is questionable
- *Automation vs. Human intervention*: Some verification may require human involvement

Solution: Establish full-chain verifiable data lineage. All data must have verifiable sources, maintain numbered traceability throughout, undergo layered verification in independent contexts, and unqualified data must be firmly eliminated.

Core principles:

1. **Data must have verifiable sources**: Every collected data item must have a URL or academic citation. Sources must be externally accessible and verifiable. Do not accept vague sourcing like "it is well known" or "reportedly."
2. **Full-chain numbering and tracing**: Raw materials have unique identifiers. Analysis materials cite raw material identifiers. Final report references point to verifiable sources.
3. **Independent context verification**: Data verification must be performed by a separate subagent. The verifier agent does not share context with the collector agent. This enables effective hallucination detection.
4. **Layered verification mechanism**: After collection—fact-checking (does the data actually exist?); After analysis—argument verification (do citations accurately support conclusions?)
5. **Unqualified data handling**: Untruthful data—firmly eliminate. Low-credibility sources—downgrade or eliminate. Uncertain credibility—clearly mark.

The data lineage chain:

```
External Source (URL/Academic Citation)
    |
    v
Raw Material [ID: SRC-001]
    | ← Post-collection verification: verify URL valid, content matches
    v
Analysis Material [ID: ANA-001, cites: SRC-001, SRC-003]
    | ← Post-analysis verification: verify citations accurate, arguments valid
    v
Final Report [References: URL list]
    |
    v
Externally Verifiable
```

Example: The industry analysis system includes a dedicated Data Verifier agent (`03.data_verifier.md`) that runs after deep research completes:

```
# Data Verifier

## Your Role

You are an independent data verifier. Your task is to verify the
authenticity and credibility of collected data.

## Critical Principle

You must independently verify each data item. Do not assume prior
```

```
collection was correct. For each URL, you must actually visit and
confirm content matches. When you find problems, record them honestly-
do not "let things slide."

## Verification Content

### 1. Source Accessibility
- Visit each URL; confirm accessible
- Record inaccessible URLs

### 2. Content Matching
- Confirm URL content matches the collected summary
- Confirm quoted text actually exists in source
- Mark data items with mismatched content

### 3. Source Credibility
- Assess source authority (official > mainstream media > industry > personal)
- Mark low-credibility sources
```

The critical design choice is that verification runs in an **independent context**. A fresh subagent that did not perform the collection has no "memory" of what the data "should" be. It approaches verification without the cognitive biases that would prevent the original collector from recognizing its own hallucinations.

Discussion: Verifiable data lineage imposes significant overhead. The verification stage may take as long as the collection stage itself, as each URL must be re-visited and content re-examined. For the industry analysis system's 55 functional items, each with multiple sources, verification represents substantial runtime.

This cost is justified for outputs requiring high credibility. The eVTOL analysis report, with its 150+ cited sources, would be worthless if sources proved fictitious upon inspection. The verification stage caught and eliminated approximately 8-12% of initially collected data in test runs—a significant proportion that would have contaminated subsequent analysis.

For systems with lower credibility requirements, this pattern may be relaxed. But for research-oriented MAS producing analytical outputs that humans will rely upon, verifiable data lineage is essential.

5. Applying the Pattern Language

Patterns gain meaning through combination. This section demonstrates how the eight essential patterns combine to form working systems, using the industry analysis system as concrete illustration.

5.1 Minimal Viable Configuration

The absolute minimum for a declarative MAS requires all four Must patterns:

Pattern	Necessity	Role
COR-02: Intelligent Runtime	Foundation	Provides execution capability
COR-01: Prompt-Defined Agent	Definition	Enables agent specification
BHV-02: Faithful Agent Instantiation	Execution	Ensures correct invocation
QUA-03: Verifiable Data Lineage	Trust	Prevents hallucination contamination

With only these four patterns, you can build a single-agent system that:

- Defines its behavior through a natural language blueprint
- Executes on an intelligent runtime
- Produces outputs with verifiable sources

However, such a minimal system lacks structure for multi-agent coordination, clear data organization, or systematic quality assurance beyond source verification.

5.2 Recommended Configuration

Adding the four recommended patterns covered in Section 4 creates a robust multi-agent system:

Pattern	Addition	Benefit
STR-01: Reference Data Configuration	Knowledge	Separates domain knowledge from agent logic
BHV-01: Orchestrated Agent Pipeline	Coordination	Enables multi-stage agent collaboration
STR-02: Filesystem Data Bus	Communication	Provides transparent inter-agent data passing
STR-03: Workspace Isolation	Safety	Prevents cross-project contamination

This eight-pattern configuration—the set presented in Section 4—supports systems of moderate complexity with multiple agents, clear data flow, and verifiable outputs.

5.3 The Industry Analysis System Configuration

The complete industry analysis system employs 12 of the 15 patterns:

Must (all 4):

- COR-01: Prompt-Defined Agent
- COR-02: Intelligent Runtime
- BHV-02: Faithful Agent Instantiation
- QUA-03: Verifiable Data Lineage

Recommended (7 of 8):

- STR-01: Reference Data Configuration
- STR-02: Filesystem Data Bus
- STR-03: Workspace Isolation
- STR-04: Business-Driven Agent Design
- STR-06: Methodological Guidance
- BHV-01: Orchestrated Agent Pipeline
- QUA-01: Embedded Quality Standards

Optional (1 of 3):

- BHV-03: Parallel Instance Execution

The system omits STR-05 (Composable Document Assembly) because the final report is generated as a unified document rather than assembled from independent chapters. It omits BHV-04 (Progressive Data Refinement) because the refinement pattern is implicit in the stage-based pipeline. It omits QUA-02 (Layered Quality Assurance) because verification occurs at specific pipeline stages rather than across multiple quality layers.

5.4 Pattern Interaction in Practice

To illustrate how patterns interact concretely, consider the execution of Stage 2 (Deep Research) in the industry analysis system:

1. Reference Data Configuration (STR-01) + Methodological Guidance (STR-06)

Before research begins, the Deep Researcher agent reads:

- `references/methodology/data-sources.md` — which sources to prioritize
- `references/methodology/analysis-methods.md` — what information to collect
- `references/domain/theoretical_framework_explained.md` — what the functional item means

These reference files tell the agent what to look for without hardcoding URLs or search strategies into the blueprint.

2. Orchestrated Agent Pipeline (BHV-01) + Faithful Agent Instantiation (BHV-02)

The orchestrator launches 55 independent Deep Researcher subagents, one per functional item:

```
For FUNCTION_ID in [1.1, 1.2, ..., 4.12]:
    Launch subagent:
        - Read agents/02.deep_researcher.md
        - Execute with parameters: INDUSTRY_ID=evtol, FUNCTION_ID={id}
    # Do NOT summarize the blueprint
    # Do NOT bundle multiple items
```

Each subagent reads the complete blueprint and executes independently. The orchestrator passes only parameters.

3. Parallel Instance Execution (BHV-03)

The 55 research tasks are independent—item 1.1's research does not depend on item 1.2's results. The orchestrator can launch all 55 subagents simultaneously:

```
Launch in parallel:
    - Subagent 1: FUNCTION_ID=1.1
    - Subagent 2: FUNCTION_ID=1.2
    ...
    - Subagent 55: FUNCTION_ID=4.12

Wait for all to complete
```

Parallel execution reduces a 55× sequential runtime to approximately 10× runtime (limited by concurrent execution capacity).

4. Filesystem Data Bus (STR-02) + Workspace Isolation (STR-03)

Each subagent writes to its designated directory:

```
data/evtol/01.materials/03.deep_research/
+-- 1.1_control_strategic_sectors/
|   +-- policies.md
|   +-- statistics.md
|   +-- cases.md
|   +-- expert_views.md
|   `-- source_list.md
+-- 1.2_provide_public_goods/
|   `-- ...
...
```

Subagents write only within `data/evtol/` (workspace isolation). The directory structure reflects data organization (filesystem data bus). Parallel subagents write to different directories, avoiding conflicts.

5. Verifiable Data Lineage (QUA-03)

After all 55 research tasks complete, Stage 3 launches the Data Verifier:

```
Launch subagent (fresh context):
    - Read agents/03.data_verifier.md
    - Verify all data in data/evtol/01.materials/03.deep_research/
```

The verifier runs in a fresh context without access to the research agents' "memories." It visits each URL, confirms content matches, and flags problems. Unverifiable data is eliminated before analysis.

6. Embedded Quality Standards (QUA-01)

Quality standards are embedded in each blueprint. The Deep Researcher blueprint includes:

```
## Quality Requirements

### Source Requirements
- Every data item must have a URL
- Prioritize authoritative sources
- Cross-verify important claims

### Completion Criteria
- [ ] Policies: At least 3 relevant policies identified
- [ ] Statistics: Quantitative data with sources
- [ ] Cases: At least 2 concrete examples
- [ ] Expert Views: At least 2 expert opinions
- [ ] Source List: All sources documented with URLs
```

The agent self-checks against these criteria before reporting completion. The orchestrator validates against the same criteria when receiving results.

5.5 The Complete Flow

Combining all patterns, a complete industry analysis run proceeds:

```
Orchestrator reads:
    - agents/00.orchestrator.md (its own blueprint)
    - references/methodology/research-overview.md (what to do)

Stage 0: Initialize
    - Verify references/ exists and is complete
    - Create data/evtol/ directory structure

Stage 1: Initial Scanning
    - Launch Initial Scanner (reads own blueprint + references)
    - Produces: industry overview, question lists
    - Orchestrator validates against completion criteria

Stage 2: Deep Research (parallel)
    - Launch 55 Deep Researcher instances
    - Each reads own blueprint + references
    - Each produces: policies, statistics, cases, expert views, source list
    - Orchestrator validates each against completion criteria

Stage 3: Data Verification (independent context)
    - Launch Data Verifier in fresh context
    - Visits every URL, confirms content
    - Eliminates unverifiable data
    - Orchestrator reviews verification report

Stage 4: Analysis (parallel)
    - Launch 55 Analyzer instances
    - Each reads verified research + theoretical framework
    - Each produces: functional item analysis
    - Orchestrator validates against completion criteria

Stage 5: Synthesis
    - Launch Synthesizer
    - Reads all functional analyses
    - Produces: feature syntheses, dimension syntheses, overall synthesis

Stage 6: Report Generation
    - Launch Reporter
    - Reads syntheses, follows output template
    - Produces: final_report.md, executive_summary.md

Stage 7: Quality Review
    - Launch Quality Checker
    - Reviews entire output against standards
    - Produces: quality_review.md
```

```
Complete
```

The pattern language provides the vocabulary to describe this flow. Each stage employs specific patterns; the interaction between patterns creates the complete system behavior.

6. The Key Insight: Executable Pattern Languages

Beyond documenting patterns for declarative MAS, this work advances a broader insight about the nature of pattern languages in the AI era: pattern languages themselves become executable. This section elaborates this insight and its implications.

6.1 Traditional Pattern Languages vs. Executable Pattern Languages

Christopher Alexander's original pattern language for architecture and the Gang of Four's design patterns for software share a common characteristic: patterns serve as design guidance that humans must translate into concrete implementations. A developer reads the Singleton pattern, understands its intent and structure, then writes code that embodies the pattern. The pattern is not directly executable—it requires human interpretation and implementation.

This characteristic persists across pattern literature. Patterns are documentation for human consumption. They accelerate learning and enable communication, but the gap between pattern description and working system must be bridged by human effort.

POMASA operates differently. Because the intelligent runtime (COR-02) can interpret natural language descriptions, and because agent blueprints (COR-01) are themselves natural language, the pattern descriptions in this paper are not merely design guidance—they are sufficiently precise that an AI system can apply them directly.

Aspect	Traditional Patterns	Executable Patterns
Consumer	Human developers	Human developers and AI systems
Implementation	Human writes code based on pattern	AI generates system based on patterns
Gap to working system	Requires human translation	Minimal or none
Precision required	Sufficient for human understanding	Sufficient for AI interpretation

6.2 The Generator: One Way to Execute the Pattern Language

POMASA includes a generator—a prompt that guides an AI system in creating new MAS based on the pattern language. The generator is not a code generator in the traditional sense; it is a prompt that tells the AI:

1. Read the user's requirements (research topic, methodology, quality expectations)
2. Select appropriate patterns from POMASA based on requirements
3. Generate agent blueprints that implement selected patterns
4. Create reference data structure
5. Produce a complete, runnable system

The generator works because:

- The AI understands natural language pattern descriptions
- The AI can apply patterns to new domains
- The AI can generate natural language blueprints that implement patterns
- The resulting blueprints are executable on the same or similar AI runtime

Crucially, **the generator is not the only way to execute the pattern language**. Any practitioner can read this paper's pattern descriptions and:

- Ask an AI to help them build a system following these patterns
- Manually write blueprints that embody the patterns
- Use the patterns as checklists while developing their own approach

The generator is merely a convenience—one worked example of pattern language execution.

6.3 Implications for Open Source

Traditional open source means publishing source code. Developers download code, study it, modify it, and run it. The source code is the artifact that enables replication and extension.

For executable pattern languages, **publishing the pattern language itself is equivalent to open source**. This paper contains sufficient information for readers to reconstruct POMASA-compliant systems:

- Pattern descriptions explain what each pattern does and why
- Examples show how patterns manifest in concrete blueprints
- The structure section explains how patterns relate
- The application section demonstrates pattern combination

A reader with access to an intelligent runtime (Claude Code or equivalent) can present this paper's content to the AI and request: "Build me a research system following these patterns for domain X." The AI can generate a working system. No traditional source code repository is required.

This shifts what "open source" means in the AI era:

- **Traditional**: Publish code $\rightarrow$ others run the code
- **AI era**: Publish patterns $\rightarrow$ others regenerate equivalent systems

The implications are significant:

1. **Knowledge transfer**: Patterns transfer more readily than code. Code embodies patterns but obscures them in implementation details. Patterns are the transferable knowledge; code is one reification.

2. **Adaptation**: Regenerating a system for a new domain may be easier than modifying existing code. The AI adapts patterns to context rather than developers adapting code.

3. **Versioning**: Pattern language evolution differs from code versioning. Patterns can evolve in description while remaining implementable; generated systems reflect current pattern understanding.

6.4 Comparison with Imperative Generation

To sharpen the distinction, consider an alternative approach: imperative generation tools that produce MAS through step-by-step instructions.

One of the authors previously developed AgentForge, an imperative generator for multi-agent systems. AgentForge's `generator.md` contains instructions like:

```
1. Create a directory named `agents/`
2. For each stage in the pipeline:
   a. Create a file `{stage_number}.{stage_name}.md`
   b. Write the following structure:
      - Role section describing the agent's purpose
      - Input section listing parameters
      - Process section with numbered steps
      - Output section specifying file locations
3. Create a directory named `references/`
...
```

This imperative approach works but has limitations:

Extensibility: Adding new capabilities requires modifying the generator instructions. Users cannot easily add "verification" or "parallel execution" without understanding and editing the generation logic.

Customization: Users who want some features but not others must edit the generator—a programming task. The generator doesn't support declarative feature selection.

Transparency: The generator's logic is opaque. Users see inputs and outputs but not the reasoning connecting them.

POMASA's pattern language approach addresses these limitations:

Extensibility: Adding capabilities means adding patterns. Users select which patterns to apply. The pattern language grows by accretion, not modification.

Customization: Users declare which patterns their system should use. An AI can generate systems with different pattern combinations without modifying the pattern language itself.

Transparency: Patterns are transparent design decisions. Users understand what each pattern contributes and can reason about trade-offs.

The fundamental difference: imperative generators tell AI "how to generate"; pattern languages tell AI "what properties the result should have." The AI figures out how to achieve properties—the same declarative-over-imperative shift that defines POMASA's approach to agents themselves.

6.5 Why Pattern Languages Suit AI Execution

Pattern languages have properties that make them particularly suitable for AI interpretation:

Natural language: Patterns are traditionally written for human readers in natural language. This is precisely what LLMs process well. No translation layer is needed.

Contextual: Patterns include context descriptions—when to apply them, what forces they balance. AI systems can match contexts to user requirements.

Compositional: Patterns are designed to combine. Pattern languages describe how patterns relate and reinforce each other. This compositional structure guides AI in assembling coherent systems.

Intent-focused: Patterns describe intent and rationale, not just structure. AI systems that understand "why" can make better implementation decisions than systems following mechanical rules.

Example-rich: Pattern literature traditionally includes examples. Examples provide grounding that helps AI systems generalize patterns to new contexts.

These characteristics emerged from patterns being designed for human learning. They turn out to be equally valuable for AI interpretation—a fortunate alignment that makes pattern languages a natural fit for the AI era.

6.6 Limitations of Executable Pattern Languages

The approach has limitations:

Runtime dependency: Executable patterns require capable AI runtimes. As of this writing, only a few systems (notably Claude Code) provide sufficient capability. Pattern languages that depend on advanced reasoning may not execute reliably on all systems.

Non-determinism: AI interpretation introduces variability. The same patterns may generate somewhat different systems across runs. For applications requiring exact replication, this variability is problematic.

Verification challenges: Verifying that a generated system correctly implements patterns is harder than verifying code correctness. The AI's interpretation may subtly deviate from pattern intent.

Capability ceiling: Current AI systems have reasoning limitations. Highly complex pattern combinations may exceed what AI can reliably implement. The pattern language's effective scope is bounded by AI capability.

Despite these limitations, the executable pattern language approach offers enough value—in knowledge transfer, adaptation ease, and conceptual clarity—to merit serious consideration for declarative MAS development.

7. Discussion and Conclusion

7.1 Applicability and Scope

POMASA emerged from building research-oriented multi-agent systems—specifically, systems that follow the pattern of "information collection → information processing → report generation." This pattern characterizes a broad class of knowledge work:

- Industry analysis and market research
- Literature reviews and systematic surveys
- Competitive intelligence gathering
- Policy analysis and impact assessment
- Due diligence and investigation reports

The industry analysis system demonstrates POMASA's applicability to substantive research tasks. The eVTOL analysis report—comprehensive, multi-dimensional, grounded in verifiable sources—suggests that declarative MAS can produce outputs of genuine analytical value.

POMASA's applicability extends beyond its original domain. The pattern language addresses general challenges in multi-agent coordination: how to define agents, how they communicate, how to ensure quality. These challenges arise regardless of whether agents collect industry data or perform other tasks.

The pattern language also accommodates extension. Systems requiring data from structured databases (rather than web search) can add patterns for database access through tools like MCP (Model Context Protocol). The filesystem data bus pattern doesn't preclude other data sources; it addresses inter-agent communication, not data origination.

7.2 Limitations

Several limitations constrain POMASA's current scope:

Batch processing orientation: The patterns assume batch processing rather than real-time interaction. Systems requiring immediate responses or continuous operation need additional patterns not yet developed.

Research task bias: The pattern language reflects its origin in research systems. Applications like code generation, customer service, or creative writing may require different or additional patterns.

Single runtime assumption: Current patterns assume a single intelligent runtime type (Claude Code). Multi-runtime systems or systems spanning different AI platforms need patterns for cross-platform coordination.

Quality ceiling: Output quality depends on underlying AI capabilities. POMASA provides structure and verification but cannot exceed what the AI runtime can produce. As AI capabilities evolve, achievable quality will increase.

Scaling limits: The industry analysis system handles 55 parallel research tasks comfortably. Systems requiring thousands of concurrent agents may encounter coordination challenges the current patterns don't address.

7.3 Pattern Language Evolution

POMASA continues to evolve. The pattern catalog grew from 5 patterns in early versions to 15 patterns currently. Growth follows a pattern:

1. **Build systems**: Construct working MAS for real tasks
2. **Encounter problems**: Identify recurring challenges
3. **Extract patterns**: Abstract solutions that work across contexts
4. **Validate patterns**: Apply extracted patterns to new systems
5. **Document patterns**: Write patterns in standard format

The BHV-02 (Faithful Agent Instantiation) pattern illustrates this evolution. Early system versions suffered quality degradation in long call chains. Investigation revealed orchestrators were summarizing blueprints rather than having subagents read them directly. The pattern crystallized from debugging this failure mode. Later, production incidents revealed that subagents sometimes unilaterally lowered quality standards when facing difficulties—prompting the addition of "completion criteria are non-negotiable" to the pattern.

Patterns emerge from practice, not theory. POMASA captures what we've learned building declarative MAS. As more systems are built and more failure modes discovered, the pattern language will continue growing.

7.4 Contributions

This paper makes three contributions to the emerging field of declarative multi-agent systems:

A pattern language: POMASA provides 15 patterns organized by category (Core, Structure, Behavior, Quality) and necessity (Must, Recommended, Optional). The patterns address foundational challenges in declarative MAS: agent definition, runtime execution, inter-agent communication, coordination, and quality assurance.

A case study: The industry analysis system demonstrates patterns in practice. The system's outputs—including a 50,000-word analysis of the eVTOL industry with 150+ verifiable sources—validate that pattern-based construction produces substantive results.

An insight about executability: The paper argues that pattern languages become executable in the AI era. Unlike traditional patterns that require human translation into code, POMASA patterns can be interpreted by AI systems to generate working implementations. This property transforms what "open source" means: publishing patterns equals publishing source.

7.5 Conclusion

The emergence of AI systems capable of understanding and executing natural language opens new possibilities for software architecture. Declarative multi-agent systems—where agents are defined by what they should achieve rather than how they should operate—represent one such possibility.

POMASA provides vocabulary and guidance for this emerging paradigm. The pattern language captures recurring solutions to common challenges, enabling practitioners to build on accumulated wisdom rather than rediscovering solutions independently.

More fundamentally, the executability of pattern languages suggests a shift in how architectural knowledge propagates. When patterns can be directly interpreted by AI systems, the distinction between "design documentation" and "source code" blurs. Architectural knowledge becomes directly actionable.

We offer POMASA as a contribution to this evolving field. The patterns are not final answers but current best understanding—understanding that will deepen as more systems are built, more failure modes discovered, and more solutions extracted into patterns. The pattern language is open for extension by the community of practitioners working in this space.

The AI era demands new thinking about software development. Pattern languages, refined over decades for human knowledge transfer, prove remarkably suitable for AI interpretation as well. POMASA demonstrates one way to harness this alignment, enabling the construction of sophisticated multi-agent systems through declarative specification and pattern-guided design.