Praval: The Complete Manual

Building Multi-Agent AI Systems from First Principles

Rajesh Sampathkumar (@aiexplorations)

Version 0.7.11 | November 2024

Contents

Chapter 1

PART I: FUNDAMENTALS

1.1 Chapter 1: Philosophy & Core Concepts

1.1.1 The Monolith Problem

It's 2:47 AM, and I'm staring at an AI agent that's become unmaintainable. Started simple: 50 lines to analyze business ideas. But then came feature requests: market research, competitive analysis, financial projections, SWOT analysis, PDF reports. Each reasonable alone. Together? 847 lines of complexity, nested conditionals, and debugging by archaeology.

This is the monolith problem. Monolithic agents force you to cram everything into one entity. They're responsible for multiple domains, maintain complex state, make cascading decisions. Your brain can hold ~7 items in working memory. When code exceeds that, you stop understanding and start guessing.

But here's the deeper problem: **monoliths prevent good thinking.** When one agent does everything, you can't think clearly about any one thing.

1.1.2 What Nature Already Knows

A coral polyp is absurdly simple. It filters nutrients and builds calcium carbonate. That's it. No complex decision-making. Just a specialist doing one job.

But thousands of these specialists create reef ecosystems so complex they sustain entire marine biomes. **The polyps don't know they're building a reef.** They're doing their thing, broadcasting outputs, consuming inputs. System-level intelligence emerges from interaction, not from individual sophistication.

That's Praval. Instead of one massive agent trying everything, imagine specialists: A researcher. An analyst. A writer. Each simple enough to understand completely. Each

excellent at their specialty. Together, through clear communication, they create intelligence that feels almost magical.

1.1.3 Emergence: Intelligence from Collaboration

Praval agents don't orchestrate complexity—they enable emergence.

Traditional approach: Central controller calls agent A, then agent B, then agent C. Carefully orchestrated. Tightly coupled. Fragile.

Praval approach: Agents defined by identity, not instructions. They broadcast findings. Respond to relevant messages. Coordinate naturally based on message types.

```
from praval import agent, chat, broadcast, start_agents

@agent("researcher", responds_to=["research_query"])
def research_agent(spore):
    """I find and analyze information."""
    findings = chat(f"Research this deeply: {spore.knowledge['query']}")
    broadcast({"type": "research_complete", "findings": findings})
    return {"research": findings}

@agent("analyst", responds_to=["research_complete"])
def analyst_agent(spore):
    """I identify patterns and insights."""
    analysis = chat(f"Analyze: {spore.knowledge['findings']}")
    return {"analysis": analysis}
```

Neither agent knows the other exists. This is emergence: system-level intelligence from agent-level simplicity.

1.1.4 Core Principles

1. **Specialization Over Generalization**: Each agent excels at one thing
2. **Declarative Design**: Define identities and patterns, let behavior emerge
3. **Knowledge-First Communication**: All communication carries structured data
4. **Composability**: Agents combine naturally
5. **Emergence**: Intelligence from collaboration, not sophistication

---

1.2 Chapter 2: Getting Started

1.2.1 Installation

```
# Basic (recommended for getting started)
pip install praval

# With memory capabilities
pip install praval[memory]

# Full installation (all features)
pip install praval[all]
```

1.2.2 Environment Setup

Praval requires at least one LLM provider API key. You can use OpenAI, Anthropic, or Cohere. The framework automatically selects the first available provider based on your environment variables.

Create a `.env` file in your project root:

```
OPENAI_API_KEY=sk-...
PRAVAL_DEFAULT_MODEL=gpt-4o-mini
```

Alternatively, you can export these as environment variables in your shell.

1.2.3 Your First Agent

Let's create a simple agent that demonstrates the core concepts. This agent will greet users by name, using an LLM to generate personalized greetings. Notice how the `@agent()` decorator transforms a regular Python function into an intelligent agent that responds to specific message types.

```
from praval import agent, chat, start_agents

@agent("greeter", responds_to=["greeting"])
def greeting_agent(spore):
    """I'm a friendly agent that greets users warmly."""
    name = spore.knowledge.get("name", "friend")
    greeting = chat(f"Generate a warm greeting for {name}")
    print(f"Agent says: {greeting}")
    return {"greeting": greeting}
```

```
start_agents(greeting_agent,
    initial_data={"type": "greeting", "name": "Alice"})
```

When you run this, Praval initializes the agent system, sends the greeting message, and the agent processes it. The `chat()` function handles the LLM interaction automatically, using your configured provider and model.

1.2.4 Multi-Agent Collaboration

Now let's see the real power of Praval: multiple agents working together through message passing. In this example, three specialized agents create a research pipeline. The researcher gathers information, the analyst extracts insights, and the writer creates a polished report. Notice that no agent explicitly calls another—they coordinate purely through broadcasting and responding to message types.

```
from praval import agent, chat, broadcast, start_agents

@agent("researcher", responds_to=["research_request"])
def research_agent(spore):
    """I research topics thoroughly."""
    findings = chat(f"Research: {spore.knowledge['topic']}")
    broadcast({"type": "analysis_request", "findings": findings})
    return {"findings": findings}

@agent("analyst", responds_to=["analysis_request"])
def analysis_agent(spore):
    """I analyze research for insights."""
    analysis = chat(f"Analyze: {spore.knowledge['findings']}")
    broadcast({"type": "report_request", "analysis": analysis})
    return {"analysis": analysis}

@agent("writer", responds_to=["report_request"])
def report_agent(spore):
    """I create clear reports."""
    report = chat(f"Write report: {spore.knowledge['analysis']}")
    print(f"\n Report:\n{report}\n")
    return {"report": report}

# They coordinate automatically!
start_agents(researcher, analyst, report_agent,
    initial_data={"type": "research_request", "topic": "quantum computing"})
```

This creates an automatic pipeline: researcher → analyst → writer. Each agent broadcasts its results with a specific message type, triggering the next agent in the chain. This loose coupling makes the system resilient and easy to extend.

1.3 Chapter 3: Building Multi-Agent Systems

1.3.1 Design Patterns

Understanding common multi-agent patterns helps you design robust systems. These patterns have emerged from real-world Praval applications and solve specific architectural challenges. Choose the pattern that matches your problem structure.

1.3.1.1 Linear Pipeline

The simplest pattern: data flows through a sequence of transformations. Each agent performs one step and passes results to the next. This works well for ETL (Extract, Transform, Load) workflows, data processing pipelines, and any task with clear sequential stages.

```
@agent("collector", responds_to=["start"])
def collect_data(spore):
    data = fetch_data(spore.knowledge["source"])
    broadcast({"type": "clean", "raw": data})

@agent("cleaner", responds_to=["clean"])
def clean_data(spore):
    cleaned = process(spore.knowledge["raw"])
    broadcast({"type": "analyze", "data": cleaned})

@agent("analyzer", responds_to=["analyze"])
def analyze_data(spore):
    insights = analyze(spore.knowledge["data"])
    return {"insights": insights}
```

Each agent in the pipeline waits for its trigger message type, processes the data, and broadcasts the result. If any agent fails, the others continue independently—the system degrades gracefully.

1.3.1.2 Parallel Processing

When you need to analyze the same input from multiple perspectives simultaneously, use parallel processing. This pattern excels at breaking down complex analysis into independent subtasks that can run concurrently, then aggregating the results. It dramatically reduces latency for multi-faceted analysis.

```
@agent("research", responds_to=["analyze_company"])
def research(spore):
    result = deep_research(spore.knowledge["company"])
    broadcast({"type": "research_done", "result": result})

@agent("financial", responds_to=["analyze_company"])
def financial(spore):
    result = analyze_financials(spore.knowledge["company"])
    broadcast({"type": "financial_done", "result": result})

@agent("aggregator", responds_to=["research_done", "financial_done"])
def aggregate(spore):
    # Combine all results
    pass
```

All three specialist agents respond to the same initial message type (`"analyze_company"`) and work simultaneously. The aggregator waits for all three completion messages before proceeding. This pattern is perfect for gathering multiple data sources or running independent analyses.

1.3.1.3 Hierarchical Routing

Sometimes you need different processing logic based on input characteristics. The hierarchical pattern uses a classifier agent to route requests to specialized handlers. This keeps specialist agents focused and avoids complex conditional logic within individual agents.

```
@agent("classifier", responds_to=["request"])
def classify(spore):
    req_type = classify_request(spore.knowledge["request"])
    broadcast({"type": f"{req_type}_request", "data": spore.knowledge})

@agent("tech", responds_to=["technical_request"])
def handle_technical(spore):
    # Handle technical questions
    pass
```

```
@agent("business", responds_to=["business_request"])
def handle_business(spore):
    # Handle business questions
    pass
```

The classifier examines each request and determines its type, then broadcasts to the appropriate channel. Each specialist only handles requests in its domain. This pattern scales well—adding a new request type just means adding a new specialist agent.

1.3.2 Real-World Example: Business Analyzer

Let's see all these patterns in action with a complete application. VentureLens is a business idea analyzer that conducts interviews, researches markets, evaluates viability, and generates professional reports. It demonstrates the Linear Pipeline pattern (interview → research → analysis → reporting) with dynamic, AI-driven question generation. This is the flagship example that reduced 847 lines of monolithic code to 225 lines of clean, composable agents.

```
venture = {}

@agent("interviewer", responds_to=["start", "answer"])
def interview(spore):
    """I conduct interviews with dynamic questions."""
    if venture.get("questions_asked", 0) < 8:
        question = chat(f"Generate next question. Context: {venture.get('responses', [])}")
        print(f"\n {question}")
        answer = input("Answer: ")
        venture.setdefault("responses", []).append({"q": question, "a": answer})
        venture["questions_asked"] = venture.get("questions_asked", 0) + 1
        broadcast({"type": "answer"})
    else:
        broadcast({"type": "interview_done", "responses": venture["responses"]})

@agent("researcher", responds_to=["interview_done"])
def research(spore):
    """I research market conditions."""
    research = chat(f"Research market for: {spore.knowledge['responses']}")
    broadcast({"type": "research_done", "research": research})

@agent("analyst", responds_to=["research_done"])
def analyze(spore):
```

```
    """I evaluate viability across 6 dimensions."""
    analysis = chat(f"""Analyze across:
    1. Problem-Solution Fit (1-10)
    2. Market Potential (1-10)
    3. Competitive Advantage (1-10)
    4. Execution Feasibility (1-10)
    5. Financial Viability (1-10)
    6. Scalability (1-10)
    Data: {venture['responses']}, {spore.knowledge['research']}""")
    broadcast({"type": "analysis_done", "analysis": analysis})

@agent("reporter", responds_to=["analysis_done"])
def report(spore):
    """I create professional reports."""
    report = chat(f"Create markdown report with sections: Executive Summary, Analysis, Recom
    with open("analysis.md", "w") as f:
        f.write(report)
    print("\n  Report saved: analysis.md")
```

Result: 847 lines → 225 lines. Each agent understandable. System more capable.

1.4 Chapter 3.5: Thinking in Praval

1.4.1 Mental Models: The Fundamental Shift

1.4.1.1 Emergence vs Orchestration

Traditional programming teaches us to orchestrate: "Do step 1, then step 2, then step 3." This works for deterministic systems but fights the nature of AI agents.

Praval teaches emergence: "Here are specialists. They communicate. Intelligence arises."

The mindset shift:

Orchestration thinking (traditional):

```
# Central controller manages everything
def analyze_business(idea):
    interview_data = interviewer.run(idea)      # Step 1
    research_data = researcher.run(interview)   # Step 2
    analysis = analyst.run(research)            # Step 3
```

```
    report = reporter.run(analysis)              # Step 4
    return report
```

Emergence thinking (Praval):

```
# Agents coordinate through messages
@agent("interviewer", responds_to=["start"])
def interview(spore):
    data = conduct_interview()
    broadcast({"type": "interview_done", "data": data})

@agent("researcher", responds_to=["interview_done"])
def research(spore):
    findings = research_market(spore.knowledge["data"])
    broadcast({"type": "research_done", "findings": findings})

# No central coordinator. Each agent knows its role.
```

Why emergence wins:

- **Resilience**: One agent failing doesn't break the system
- **Extensibility**: Add agents without touching existing code
- **Parallelism**: Agents naturally run concurrently
- **Clarity**: Each agent is simple enough to hold in your head

1.4.1.2 Identity Drives Behavior

In Praval, the agent's docstring isn't documentation—it's the agent's **identity**.

```
@agent("philosopher")
def philosophical_agent(spore):
    """I think deeply about questions, exploring different perspectives
    through existentialist, pragmatic, and stoic lenses."""

    # This agent doesn't follow instructions
    # It acts according to its nature
```

The LLM reads this identity and becomes that agent. This enables:

- **Adaptive behavior**: Agent handles novel situations naturally
- **Consistent persona**: Same identity across all interactions
- **Emergent capabilities**: Identity enables behaviors you didn't explicitly code

The "I am" test: If you can't complete "I am a _____ that _____", your agent needs

clearer identity.

1.4.1.3 The Cognitive Load Test: When to Split Agents

How do you know when one agent should become two?

The 7±2 Rule: If you can't hold the agent's full responsibility in working memory, split it.

Signals it's time to split:

- **Length**: Agent exceeds 200 lines
- **Responsibilities**: Agent handles more than 3 distinct concerns
- **Message types**: Agent responds to more than 5 different message types
- **Conditional complexity**: Deep nested if/else based on message type
- **Testing difficulty**: Can't test the agent without complex mocking

Example - before split:

```
@agent("assistant", responds_to=["question", "search", "analyze", "summarize"])
def do_everything(spore):
    """I handle questions, search, analysis, and summarization."""
    msg_type = spore.knowledge["type"]

    if msg_type == "question":
        # 50 lines of question handling
    elif msg_type == "search":
        # 60 lines of search logic
    elif msg_type == "analyze":
        # 70 lines of analysis
    elif msg_type == "summarize":
        # 40 lines of summarization

    # 220 lines total, 4 responsibilities, cognitive overload
```

Example - after split:

```
@agent("qa", responds_to=["question"])
def question_answerer(spore):
    """I answer questions directly and accurately."""
    # 50 lines - single responsibility, clear

@agent("searcher", responds_to=["search"])
def search_agent(spore):
    """I search for information across sources."""
```

```
    # 60 lines - focused on search

@agent("analyst", responds_to=["analyze"])
def analysis_agent(spore):
    """I analyze data for patterns and insights."""
    # 70 lines - pure analysis

@agent("summarizer", responds_to=["summarize"])
def summary_agent(spore):
    """I create concise summaries."""
    # 40 lines - summarization only
```

Each agent now fits in working memory. Clear. Testable. Composable.

1.4.2 Anti-Patterns: Common Mistakes

1.4.2.1 The God Agent

Symptom: One agent that tries to do everything.

```
@agent("assistant", responds_to=[
    "question", "search", "analyze", "summarize", "translate",
    "code", "debug", "write", "edit", "review"
])
def handle_everything(spore):
    """I can do anything you need."""
    # 500+ lines of complexity
```

Why it fails: Violates specialization. Can't reason about it. Can't test it. Can't extend it.

Refactoring strategy:

1. **Identify clusters**: Group related message types
2. **Extract specialists**: One agent per cluster
3. **Preserve communication**: Use same message types
4. **Migrate incrementally**: One responsibility at a time

1.4.2.2 Message Storms

Symptom: Agents broadcasting too frequently, creating cascade of messages.

```
@agent("processor", responds_to=["data"])
def process(spore):
    results = []
```

```
    for item in spore.knowledge["items"]:  # 1000 items
        result = process_one(item)
        broadcast({"type": "item_done", "result": result})  # 1000 broadcasts!
```

Why it fails: Floods the reef. Overwhelms agents. Performance degrades.

Fix: Batch and aggregate:

```
@agent("processor", responds_to=["data"])
def process(spore):
    results = [process_one(item) for item in spore.knowledge["items"]]
    # Single broadcast with all results
    broadcast({"type": "batch_done", "results": results})
```

1.4.2.3 Tight Coupling Through Shared State

Symptom: Agents manipulating shared global dictionaries.

```
shared_state = {}  # Global state - danger!

@agent("agent_a")
def agent_a(spore):
    shared_state["data"] = compute_something()  # Side effect

@agent("agent_b")
def agent_b(spore):
    # Depends on agent_a having run first
    value = shared_state.get("data")  # Implicit coupling
```

Why it fails: Hidden dependencies. Race conditions. Can't reason about execution order.

Fix: Communicate through spores:

```
@agent("agent_a")
def agent_a(spore):
    data = compute_something()
    broadcast({"type": "data_ready", "data": data})  # Explicit

@agent("agent_b", responds_to=["data_ready"])
def agent_b(spore):
    # Explicitly depends on data_ready message
    value = spore.knowledge["data"]  # Clear contract
```

1.4.2.4 Premature Agent Splitting

Symptom: Creating agents before understanding the problem.

```
# Created 15 agents for a problem that needed 3
@agent("fetcher")
def fetch(): pass

@agent("parser")
def parse(): pass

@agent("validator")
def validate(): pass

@agent("transformer")
def transform(): pass

# ... 11 more agents doing trivial work
```

Why it fails: Over-engineering. Each agent adds communication overhead. Hard to follow logic.

Fix: Start simple, split when needed:

```
# Start with one agent
@agent("processor")
def process_data(spore):
    data = fetch_data()
    parsed = parse(data)
    validated = validate(parsed)
    transformed = transform(validated)
    return {"result": transformed}

# Split only when processor exceeds cognitive load
```

Rule: Start with 1 agent. Split to 2 when you feel cognitive overload. Grow to 3-5 when natural responsibilities emerge. Rarely need more than 10 agents.

1.4.3 Evolution Patterns: Growing Your System

1.4.3.1 Start Simple: 1 → 2 Agents

When: Your single agent hits 150-200 lines or has 2+ clear responsibilities.

Strategy:

1. **Identify the split point**: Where does one responsibility end and another begin?
2. **Extract the clearest responsibility**: Pull out the most independent concern
3. **Connect with messages**: Original agent broadcasts to new agent
4. **Test independently**: Ensure both agents work in isolation

Example:

Before (1 agent):

```
@agent("researcher")
def research_and_report(spore):
    """I research topics and create reports."""
    # Research (80 lines)
    findings = deep_research(spore.knowledge["topic"])

    # Report writing (70 lines)
    report = create_report(findings)

    return {"report": report}
```

After (2 agents):

```
@agent("researcher", responds_to=["research_request"])
def research_agent(spore):
    """I research topics deeply."""
    findings = deep_research(spore.knowledge["topic"])
    broadcast({"type": "findings_ready", "findings": findings})

@agent("writer", responds_to=["findings_ready"])
def report_writer(spore):
    """I create professional reports from research."""
    report = create_report(spore.knowledge["findings"])
    return {"report": report}
```

1.4.3.2 Growing: 5 → 15 Agents

When: Your system handles multiple workflows or domains.

Organization strategies:

Domain-based grouping:

```
# Customer service domain (4 agents)
@agent("classifier")  # Routes requests
@agent("qa")          # Answers questions
@agent("support")     # Technical support
@agent("escalation")  # Handles complex cases

# Analytics domain (3 agents)
@agent("collector")   # Gathers data
@agent("analyzer")    # Finds patterns
@agent("visualizer")  # Creates charts

# Content domain (3 agents)
@agent("researcher")  # Content research
@agent("writer")      # Content creation
@agent("editor")      # Quality control
```

Channel-based separation:

```
# Use channels to organize by domain
@agent("customer_qa", channel="customer_service")
@agent("tech_support", channel="customer_service")

@agent("data_analyst", channel="analytics")
@agent("viz_creator", channel="analytics")

@agent("content_writer", channel="content")
@agent("content_editor", channel="content")
```

Coordinator pattern (for very complex systems):

```
@agent("domain_coordinator", responds_to=["task"])
def coordinate(spore):
    """I route tasks to appropriate domain."""
    task_type = classify_task(spore.knowledge)

    if task_type == "customer_service":
        broadcast({"type": "customer_request", ...}, channel="customer_service")
    elif task_type == "analytics":
        broadcast({"type": "analytics_request", ...}, channel="analytics")
    elif task_type == "content":
        broadcast({"type": "content_request", ...}, channel="content")
```

1.4.3.3 Signals You Need to Split

Watch for these indicators:

Code metrics:

- Agent > 200 lines
- Cyclomatic complexity > 15
- More than 5 message types in `responds_to`
- More than 3 distinct LLM prompts in one agent

Behavioral signals:

- Testing requires extensive mocking
- Adding features means editing multiple responsibilities
- Agent name is vague ("processor", "handler", "manager")
- You can't explain agent's purpose in one sentence

Team signals:

- Multiple developers editing same agent
- Merge conflicts in agent code
- "I'm not sure what this agent does"
- Pull requests touch unrelated logic

1.4.3.4 Deprecating Agents Gracefully

Sometimes you need to retire agents:

Strategy:

1. **Mark deprecated**: Add clear deprecation notice
2. **Route to replacement**: Forward messages to new agent
3. **Monitor usage**: Check if anyone still depends on it
4. **Remove after grace period**: Delete when usage drops to zero

```
@agent("old_analyzer", responds_to=["analyze_request"])
def deprecated_analyzer(spore):
    """DEPRECATED: Use new_analyzer instead. Will be removed in v2.0."""

    # Log deprecation warning
    logger.warning("old_analyzer is deprecated, routing to new_analyzer")

    # Forward to replacement
    broadcast({
        "type": "new_analyze_request",
```

```
        "data": spore.knowledge,
        "migrated_from": "old_analyzer"
    })

@agent("new_analyzer", responds_to=["new_analyze_request"])
def new_analyzer(spore):
    """I analyze data with improved algorithms."""
    # New implementation
```

Chapter 2

PART II: CORE FEATURES

2.1 Chapter 4: Agents & Communication

2.1.1 The Decorator API

2.1.1.1 Basic Agent Creation

```
from praval import agent, chat

@agent("analyst")
def simple_agent(spore):
    """I analyze data."""
    return chat(f"Analyze: {spore.knowledge['data']}")
```

2.1.1.2 Advanced Parameters

```
@agent(
    name="researcher",
    responds_to=["research_query", "deep_dive"],
    channel="research",
    auto_broadcast=True,
    system_message="You are an expert researcher."
)
def research_agent(spore):
    """I research topics deeply."""
    result = chat(spore.knowledge["query"])
    return {"findings": result}
```

Parameters:

- `name`: Unique agent identifier
- `responds_to`: List of message types to process
- `channel`: Communication channel (default: "main")
- `auto_broadcast`: Auto-broadcast return values
- `system_message`: LLM system prompt

2.1.2 Reef Communication

2.1.2.1 Spore Structure

Spores are JSON messages carrying knowledge:

```
{
    "id": "uuid",
    "type": "research_complete",
    "knowledge": {
        "topic": "quantum computing",
        "findings": [...],
        "confidence": 0.89
    },
    "from_agent": "researcher",
    "timestamp": "2024-11-05T10:30:00Z",
    "channel": "main"
}
```

2.1.2.2 Broadcasting

```
from praval import broadcast

# Broadcast to all listeners
broadcast({
    "type": "analysis_complete",
    "insights": results,
    "confidence": 0.95
})

# Broadcast to specific channel
broadcast(data, channel="alerts")
```

2.1.2.3 Direct Messaging

```
from praval import send_to_agent

# Send to specific agent
send_to_agent(
    agent_name="analyst",
    message={"type": "urgent_analysis", "data": data}
)
```

2.1.2.4 Channels

```
from praval import get_reef

reef = get_reef()

# Create channel
reef.create_channel("research", max_capacity=500)

# Subscribe agents
@agent("researcher", channel="research")
def research_agent(spore):
    pass
```

2.1.3 Multi-LLM Support

Praval auto-selects from available providers:

```
# Set in environment
OPENAI_API_KEY=...
ANTHROPIC_API_KEY=...
COHERE_API_KEY=...

# Or configure explicitly
from praval import configure

configure({
    "default_provider": "openai",
    "default_model": "gpt-4o-mini"
})
```

Supported providers: OpenAI, Anthropic, Cohere

2.2 Chapter 5: Memory & Persistence

2.2.1 Memory Architecture

Praval provides multi-layered memory:

```
    Memory Manager

  Short-term (In-Memory)        Fast working memory
  Long-term (ChromaDB)          Semantic search
  Episodic (Conversations)      History tracking
  Semantic (Facts)              Knowledge base
```

2.2.2 Using Memory

```
from praval.memory import get_memory_manager

memory = get_memory_manager()

# Store conversation
memory.store_conversation_turn(
    agent_name="assistant",
    user_message="What is quantum computing?",
    assistant_message="Quantum computing uses qubits..."
)

# Semantic search
results = memory.search_long_term(
    query="quantum computing applications",
    top_k=5
)

# Store facts
memory.store_semantic(
    agent_name="researcher",
```

```
    fact="Quantum computers use superposition and entanglement"
)

# Get conversation history
history = memory.get_episodic_memory(agent_name="assistant")
```

2.2.3 ChromaDB Integration

Long-term memory uses embedded ChromaDB:

```
pip install praval[memory]

# Automatic setup - no configuration needed
from praval import agent, chat
from praval.memory import get_memory_manager

@agent("chatbot")
def chatbot_agent(spore):
    memory = get_memory_manager()

    # Search relevant context
    context = memory.search_long_term(spore.knowledge["query"], top_k=3)

    # Generate response with context
    response = chat(f"Context: {context}\n\nQuery: {spore.knowledge['query']}")

    # Store conversation
    memory.store_conversation_turn(
        "chatbot",
        spore.knowledge["query"],
        response
    )

    return {"response": response}
```

2.3 Chapter 6: Observability

2.3.1 Overview

Zero-configuration distributed tracing based on OpenTelemetry. Every agent execution, LLM call, and memory operation is automatically traced.

2.3.2 Key Features

Automatic Instrumentation:

```
from praval import agent, chat

@agent("researcher")
def research_agent(spore):
    # Automatically traced - no code changes!
    result = chat(f"Research: {spore.knowledge['topic']}")
    return {"findings": result}
```

View Traces:

```
from praval.observability import show_recent_traces

# Console viewer with tree display
show_recent_traces(limit=5)
```

Export to Monitoring:

```
from praval.observability import export_traces_to_otlp

# Export to Jaeger, Zipkin, Honeycomb, DataDog, etc.
export_traces_to_otlp("http://localhost:4318/v1/traces")
```

2.3.3 Configuration

```
# Enable/disable (auto-enabled in dev)
PRAVAL_OBSERVABILITY=auto  # auto | on | off

# OTLP endpoint
PRAVAL_OTLP_ENDPOINT=http://localhost:4318/v1/traces

# Sample rate (0.0-1.0)
PRAVAL_SAMPLE_RATE=1.0  # 100% sampling
```

2.3.4 What Gets Traced

- **Agent Execution**: Name, duration, inputs/outputs, errors
- **LLM Calls**: Provider, model, tokens, cost, latency
- **Memory Operations**: Type, layer, size, hit/miss
- **Reef Communication**: Message type, size, routing
- **Storage I/O**: Backend, operation, size, duration

2.3.5 Example Output

```
Trace: 7f3e8a92-4b5c-4d3e-8e9f-1a2b3c4d5e6f
Duration: 2.45s | Status: OK

  agent.researcher (2.4s)
    llm.generate (1.8s)
     Provider: openai | Model: gpt-4o-mini
     Tokens: 150/450/600 | Cost: $0.0024
    memory.search (0.3s)
     Layer: long_term | Results: 5
    reef.broadcast (0.01s)
     Type: research_complete
  agent.analyst (0.8s)
    llm.generate (0.7s)
     Tokens: 200/350/550 | Cost: $0.0022
```

2.3.6 Production Best Practices

Sampling Strategy:

```
# Production: 10% sampling
PRAVAL_SAMPLE_RATE=0.1
PRAVAL_OTLP_ENDPOINT=http://collector:4318/v1/traces
```

Performance: <5% overhead at 100% sampling, <1% at 10%

Cleanup:

```
from praval.observability import get_trace_store

store = get_trace_store()
store.cleanup_old_traces(days=7)
```

2.4 Chapter 6.5: Production Mastery

2.4.1 From Monolith to Multi-Agent: A Real Migration

This case study shows how to transform an 847-line monolithic agent into a clean multi-agent system.

2.4.1.1 The Original Monolith

```
class BusinessAnalyzer:
    """One class to rule them all."""

    def __init__(self):
        self.llm = OpenAI()
        self.questions_asked = 0
        self.responses = []
        self.research_data = None
        self.analysis = None

    def run_analysis(self, business_idea):
        # Interview phase (150 lines)
        while self.questions_asked < 8:
            question = self._generate_question()
            answer = input(question)
            self.responses.append({"q": question, "a": answer})
            self.questions_asked += 1

        # Research phase (200 lines)
        self.research_data = self._research_market()
        self.research_data.update(self._analyze_competitors())
        self.research_data.update(self._identify_trends())

        # Analysis phase (250 lines)
        self.analysis = {
            "problem_fit": self._analyze_problem_fit(),
            "market_potential": self._analyze_market(),
            "competition": self._analyze_competition(),
            "execution": self._analyze_execution(),
            "financials": self._analyze_financials(),
            "scalability": self._analyze_scalability()
        }
```

```
        # Report phase (150 lines)
        report = self._create_markdown_report()
        pdf = self._generate_pdf(report)
        self._open_in_browser(pdf)

        return report

    # ... 97 more methods doing different things
```

Problems:

- 847 lines in one file
- 101 methods across 4 responsibilities
- Can't test individual phases
- Can't reuse components
- Hard to extend or modify
- State management nightmare

2.4.1.2 Step 1: Identify Responsibilities

Map the monolith to clear domains:

Responsibility mapping:

- **Interview** (150 lines): Generate questions, collect answers
- **Research** (200 lines): Market research, competitors, trends
- **Analysis** (250 lines): Evaluate across 6 dimensions
- **Reporting** (150 lines): Create markdown, generate PDF, present

Each is independent and communicates through data.

2.4.1.3 Step 2: Extract First Agent (Interviewer)

Start with the clearest, most independent responsibility:

```
from praval import agent, chat, broadcast

venture = {}  # Temporary shared state during migration

@agent("interviewer", responds_to=["start_interview", "answer_provided"])
def interviewer_agent(spore):
    """I conduct insightful business idea interviews."""
```

```
    if venture.get("questions_asked", 0) < 8:
        # Generate contextual question
        context = venture.get("responses", [])
        question = chat(f"""Generate the next insightful question.
        Previous context: {context}
        Make it specific and actionable.""")

        print(f"\n Question {venture.get('questions_asked', 0) + 1}: {question}")
        answer = input("Your answer: ")

        venture.setdefault("responses", []).append({
            "question": question,
            "answer": answer
        })
        venture["questions_asked"] = venture.get("questions_asked", 0) + 1

        # Continue interview
        broadcast({"type": "answer_provided"})
    else:
        # Interview complete - hand off to research
        broadcast({
            "type": "interview_complete",
            "responses": venture["responses"]
        })
```

Benefits already:

- 50 lines vs 150 (simpler)
- Single responsibility (testable)
- Clear identity (understandable)

2.4.1.4 Step 3: Progressive Migration

Extract remaining responsibilities one at a time:

```
@agent("researcher", responds_to=["interview_complete"])
def researcher_agent(spore):
    """I research market conditions and competitive landscape."""

    responses = spore.knowledge["responses"]
```

```
    research = chat(f"""Based on this business idea, research:
    1. Market size and growth trends
    2. Key competitors and their positioning
    3. Market trends and opportunities

    Interview data: {responses}""")

    broadcast({
        "type": "research_complete",
        "responses": responses,
        "research": research
    })

@agent("analyst", responds_to=["research_complete"])
def analyst_agent(spore):
    """I evaluate business viability across 6 dimensions."""

    responses = spore.knowledge["responses"]
    research = spore.knowledge["research"]

    analysis = chat(f"""Analyze this business idea:

    1. Problem-Solution Fit (score 1-10)
    2. Market Potential (score 1-10)
    3. Competitive Advantage (score 1-10)
    4. Execution Feasibility (score 1-10)
    5. Financial Viability (score 1-10)
    6. Scalability (score 1-10)

    For each dimension, provide score, rationale, strengths, weaknesses.

    Interview: {responses}
    Research: {research}""")

    broadcast({
        "type": "analysis_complete",
        "analysis": analysis
    })
```

```python
@agent("reporter", responds_to=["analysis_complete"])
def reporter_agent(spore):
    """I create professional markdown reports."""

    analysis = spore.knowledge["analysis"]

    report = chat(f"""Create a professional business analysis report:

    # Executive Summary
    # Viability Scores
    # Detailed Analysis
    # Strengths & Opportunities
    # Risks & Challenges
    # Recommendations
    # Next Steps

    Analysis: {analysis}""")

    filename = "venture_analysis.md"
    with open(filename, "w") as f:
        f.write(report)

    print(f"\n  Report generated: {filename}")
```

2.4.1.5 Step 4: Results

Before (Monolith):

- 847 lines in one file
- 101 methods
- 4 tightly coupled responsibilities
- Hard to test, extend, or reuse

After (Multi-Agent):

- 225 lines total across 4 agents
- Each agent 50-70 lines
- 4 independent, composable specialists
- Easy to test, extend, and reuse

Metrics:

- **73% code reduction** (847 → 225 lines)
- **Cyclomatic complexity**: 47 → 8 (per agent)
- **Test coverage**: 12% → 94%
- **Time to add feature**: 2 hours → 20 minutes

2.4.2 Cost & Performance Optimization

2.4.2.1 Where LLM Costs Go

Real production numbers from a multi-agent RAG chatbot:

Before optimization (monthly costs):

- Research agent: $180/month (1.2M tokens)
- Analysis agent: $150/month (1M tokens)
- Response agent: $120/month (800K tokens)
- Memory search: $50/month (embedded calls)
- **Total: $500/month**

After optimization (monthly costs):

- Research agent: $60/month (400K tokens, 67% reduction)
- Analysis agent: $45/month (300K tokens, 70% reduction)
- Response agent: $30/month (200K tokens, 75% reduction)
- Memory search: $15/month (cached embeddings)
- **Total: $150/month (70% reduction)**

2.4.2.2 Optimization Strategies

Strategy 1: Aggressive Caching

```
from functools import lru_cache

# Cache expensive research calls
@lru_cache(maxsize=1000)
def cached_research(topic: str) -> str:
    """Research results cached for 24 hours."""
    return chat(f"Research: {topic}")

@agent("researcher")
def research_agent(spore):
    topic = spore.knowledge["topic"]

    # Use cache first
```

```
    result = cached_research(topic)

    broadcast({"type": "research_complete", "result": result})
```

Savings: 40-50% reduction in duplicate LLM calls

Strategy 2: Memory as Cache

```
from praval.memory import get_memory_manager

@agent("chatbot")
def chatbot_agent(spore):
    memory = get_memory_manager()
    query = spore.knowledge["query"]

    # Check if we've answered this before
    similar = memory.search_long_term(query, top_k=1)

    if similar and similar[0]["similarity"] > 0.95:
        # Return cached answer (no LLM call!)
        return {"response": similar[0]["response"]}

    # Generate new response only if needed
    response = chat(f"Answer: {query}")

    # Cache for future
    memory.store_conversation_turn("chatbot", query, response)

    return {"response": response}
```

Savings: 30-40% reduction for repeat queries

Strategy 3: Batch Processing

```
@agent("analyzer")
def batch_analyzer(spore):
    items = spore.knowledge["items"]  # 100 items

    # BAD: 100 separate LLM calls
    # results = [chat(f"Analyze: {item}") for item in items]

    # GOOD: 1 batched LLM call
```

```
    batch_prompt = f"""Analyze these items and return JSON array:
    Items: {json.dumps(items)}

    Return: [{{"item": "...", "analysis": "..."}}, ...]"""

    results = json.loads(chat(batch_prompt))

    broadcast({"type": "batch_complete", "results": results})
```

Savings: 90% reduction in API overhead

Strategy 4: Smaller Models for Simple Tasks

```
from praval import chat

@agent("classifier")
def classify_request(spore):
    # Simple classification: use cheap model
    category = chat(
        f"Classify this as technical/business/general: {spore.knowledge['text']}",
        model="gpt-4o-mini"  # $0.15/1M tokens vs $5/1M for GPT-4
    )

    broadcast({"type": f"{category}_request", "data": spore.knowledge})

@agent("analyst")
def deep_analysis(spore):
    # Complex analysis: use powerful model
    analysis = chat(
        f"Deep analysis: {spore.knowledge['data']}",
        model="gpt-4o"  # More expensive but necessary for quality
    )

    return {"analysis": analysis}
```

Savings: 60-70% cost reduction on simple tasks

Strategy 5: Rate Limiting & Queuing

```
import time
from collections import deque
```

```
request_queue = deque()
last_call_time = 0
MIN_INTERVAL = 0.1  # 10 requests/second max

@agent("researcher")
def rate_limited_research(spore):
    global last_call_time

    # Enforce rate limit
    elapsed = time.time() - last_call_time
    if elapsed < MIN_INTERVAL:
        time.sleep(MIN_INTERVAL - elapsed)

    result = chat(f"Research: {spore.knowledge['topic']}")
    last_call_time = time.time()

    return {"result": result}
```

Savings: Avoid rate limit charges, stay in tier pricing

2.4.2.3 Real Numbers: Before & After

Production RAG Chatbot (1000 daily users):

Metric	Before	After	Change
Monthly cost	$500	$150	-70%
Avg response time	3.2s	1.8s	-44%
Cache hit rate	0%	65%	+65%
Duplicate calls	42%	5%	-88%
User satisfaction	7.2/10	8.9/10	+24%

Key optimizations applied:

- Memory-based caching (30% savings)
- Batch processing (20% savings)
- Smaller models for classification (15% savings)
- Request deduplication (5% savings)

2.4.3 Testing Multi-Agent Systems

Testing emergent systems requires different strategies than testing traditional code.

2.4.3.1 Level 1: Unit Testing Individual Agents

Test agents in isolation with mock spores:

```
import pytest
from praval import agent

def test_research_agent():
    """Test researcher agent handles queries correctly."""

    results = []

    @agent("researcher", responds_to=["research_query"])
    def research_agent(spore):
        query = spore.knowledge["query"]
        result = {"findings": f"Research on {query}"}
        results.append(result)
        return result

    from praval import start_agents

    # Mock spore
    start_agents(research_agent, initial_data={
        "type": "research_query",
        "query": "quantum computing"
    })

    # Verify behavior
    assert len(results) == 1
    assert "quantum computing" in results[0]["findings"]

def test_agent_error_handling():
    """Test agent handles errors gracefully."""

    errors = []

    @agent("processor")
    def error_prone_agent(spore):
        try:
            risky_operation()
        except Exception as e:
```

```
            errors.append(str(e))
            return {"error": str(e)}

    start_agents(error_prone_agent, initial_data={"type": "process"})

    assert len(errors) == 1
```

2.4.3.2 Level 2: Integration Testing Agent Communication

Test how agents interact:

```
def test_agent_pipeline():
    """Test multi-agent pipeline communication."""

    execution_log = []

    @agent("collector", responds_to=["start"])
    def collector(spore):
        execution_log.append("collector")
        broadcast({"type": "clean", "data": "raw_data"})

    @agent("cleaner", responds_to=["clean"])
    def cleaner(spore):
        execution_log.append("cleaner")
        broadcast({"type": "analyze", "data": "clean_data"})

    @agent("analyzer", responds_to=["analyze"])
    def analyzer(spore):
        execution_log.append("analyzer")
        return {"result": "analysis_done"}

    start_agents(collector, cleaner, analyzer,
        initial_data={"type": "start"})

    # Verify execution order
    assert execution_log == ["collector", "cleaner", "analyzer"]

def test_message_filtering():
    """Test agents only respond to relevant messages."""
```

```
    executions = {"a": 0, "b": 0}

    @agent("agent_a", responds_to=["type_a"])
    def agent_a(spore):
        executions["a"] += 1

    @agent("agent_b", responds_to=["type_b"])
    def agent_b(spore):
        executions["b"] += 1

    # Send type_a message
    start_agents(agent_a, agent_b, initial_data={"type": "type_a"})

    # Only agent_a should execute
    assert executions["a"] == 1
    assert executions["b"] == 0
```

2.4.3.3 Level 3: Testing Emergent Behavior

Test system-level outcomes, not implementation:

```
def test_business_analysis_workflow():
    """Test complete business analysis produces expected results."""

    # Don't test individual agents - test final outcome

    final_report = None

    @agent("reporter", responds_to=["analysis_done"])
    def capture_report(spore):
        nonlocal final_report
        final_report = spore.knowledge["report"]

    # Run full system
    start_agents(
        interviewer, researcher, analyst, reporter, capture_report,
        initial_data={"type": "start_interview"}
    )

    # Test emergent outcomes
```

```
    assert final_report is not None
    assert "Executive Summary" in final_report
    assert "Recommendations" in final_report
    assert len(final_report) > 500  # Substantial report

def test_system_resilience():
    """Test system handles agent failures."""

    failures = []

    @agent("agent_a", responds_to=["start"])
    def failing_agent(spore):
        failures.append("agent_a_failed")
        raise Exception("Agent A failed")

    @agent("agent_b", responds_to=["start"])
    def backup_agent(spore):
        # Should still execute despite agent_a failure
        return {"status": "success"}

    # System should continue despite failure
    result = start_agents(failing_agent, backup_agent,
        initial_data={"type": "start"})

    assert len(failures) == 1
    assert result["status"] == "success"
```

2.4.3.4 Mocking Strategies

Mock LLM calls for deterministic testing:

```
from unittest.mock import patch

def test_agent_with_mocked_llm():
    """Test agent logic without real LLM calls."""

    @agent("analyzer")
    def analyzer_agent(spore):
        analysis = chat(f"Analyze: {spore.knowledge['data']}")
        return {"analysis": analysis}
```

```
    # Mock the chat function
    with patch('praval.chat') as mock_chat:
        mock_chat.return_value = "Mocked analysis result"

        result = start_agents(analyzer_agent,
            initial_data={"type": "analyze", "data": "test"})

        assert result["analysis"] == "Mocked analysis result"
        mock_chat.assert_called_once()
```

Mock spores for testing:

```
from praval.core.reef import Spore

def test_with_mock_spore():
    """Create mock spores for testing."""

    mock_spore = Spore(
        spore_type="test_type",
        knowledge={"test_key": "test_value"},
        from_agent="test_agent"
    )

    @agent("processor")
    def processor_agent(spore):
        return {"processed": spore.knowledge["test_key"]}

    # Test with mock
    result = processor_agent(mock_spore)
    assert result["processed"] == "test_value"
```

2.4.4 Advanced Communication Patterns

2.4.4.1 Request-Response with Timeouts

For critical operations that need guaranteed responses:

```
from praval import agent, broadcast
import time
import threading
```

```python
request_responses = {}

@agent("requester", responds_to=["start"])
def requester_agent(spore):
    """Request data with timeout."""
    request_id = str(uuid.uuid4())

    # Send request
    broadcast({
        "type": "data_request",
        "request_id": request_id,
        "query": "important_data"
    })

    # Wait for response with timeout
    timeout = 5.0
    start_time = time.time()

    while time.time() - start_time < timeout:
        if request_id in request_responses:
            response = request_responses[request_id]
            del request_responses[request_id]
            return {"data": response}
        time.sleep(0.1)

    # Timeout - handle gracefully
    return {"error": "Request timed out"}

@agent("responder", responds_to=["data_request"])
def responder_agent(spore):
    """Respond to data requests."""
    request_id = spore.knowledge["request_id"]

    # Process request
    data = fetch_important_data(spore.knowledge["query"])

    # Send response
    request_responses[request_id] = data
```

2.4.4.2 Circuit Breaker for Failing Agents

Prevent cascade failures when agents repeatedly fail:

```
from collections import defaultdict
import time

failure_counts = defaultdict(int)
last_failure_time = defaultdict(float)
FAILURE_THRESHOLD = 3
RESET_TIMEOUT = 60  # seconds

@agent("protected_agent", responds_to=["risky_operation"])
def circuit_breaker_agent(spore):
    """Agent with circuit breaker protection."""

    agent_name = "protected_agent"

    # Check circuit breaker
    if failure_counts[agent_name] >= FAILURE_THRESHOLD:
        time_since_failure = time.time() - last_failure_time[agent_name]

        if time_since_failure < RESET_TIMEOUT:
            # Circuit open - reject request
            return {"error": "Circuit breaker open, try again later"}
        else:
            # Reset circuit breaker
            failure_counts[agent_name] = 0

    try:
        # Attempt risky operation
        result = risky_external_api_call()

        # Success - reset failure count
        failure_counts[agent_name] = 0
        return {"result": result}

    except Exception as e:
        # Failure - increment counter
        failure_counts[agent_name] += 1
        last_failure_time[agent_name] = time.time()
```

```
        return {"error": str(e)}
```

2.4.4.3 Priority Queue for Important Messages

Handle urgent messages before routine ones:

```
import heapq
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedMessage:
    priority: int
    message: Any = field(compare=False)

priority_queue = []

@agent("urgent_classifier", responds_to=["incoming"])
def classifier_agent(spore):
    """Classify and prioritize messages."""

    # Determine priority (lower number = higher priority)
    if spore.knowledge.get("urgent"):
        priority = 1
    elif spore.knowledge.get("important"):
        priority = 2
    else:
        priority = 3

    # Add to priority queue
    heapq.heappush(priority_queue, PrioritizedMessage(
        priority=priority,
        message=spore.knowledge
    ))

    # Process highest priority message
    if priority_queue:
        next_msg = heapq.heappop(priority_queue)
        broadcast({
            "type": "process",
```

```
            "priority": next_msg.priority,
            "data": next_msg.message
        })
```

2.4.5 Memory Strategies in Production

2.4.5.1 When to Use Which Memory Layer

Short-term memory (in-memory, fast):

- Current conversation context
- Session-specific data
- Temporary calculations
- Data that doesn't need persistence

```
from praval.memory import get_memory_manager

memory = get_memory_manager()

# Store temporary context
memory.store_short_term("current_user", user_id)
memory.store_short_term("session_start", time.time())
```

Long-term memory (ChromaDB, semantic):

- Knowledge base articles
- Historical conversations
- Learned facts
- Reusable context

```
# Store permanent knowledge
memory.store_long_term(
    content="Quantum computers use qubits for parallel computation",
    metadata={"topic": "quantum_computing", "source": "research"}
)

# Semantic search
results = memory.search_long_term("how do quantum computers work", top_k=5)
```

Episodic memory (conversation history):

- User conversation history
- Agent interaction logs
- Time-ordered events

- Audit trails

```
# Store conversation
memory.store_conversation_turn(
    agent_name="assistant",
    user_message="What's the weather?",
    assistant_message="It's sunny, 72°F"
)

# Retrieve history
history = memory.get_episodic_memory("assistant", limit=10)
```

Semantic memory (facts and relationships):

- Entity relationships
- Factual knowledge
- Concept mappings
- Domain knowledge

```
# Store facts
memory.store_semantic(
    agent_name="knowledge_agent",
    fact="Python is a programming language created by Guido van Rossum"
)
```

2.4.5.2 Hybrid Search Strategy

Combine multiple memory layers for best results:

```
@agent("smart_retriever")
def hybrid_search_agent(spore):
    """Search using multiple strategies and merge results."""

    query = spore.knowledge["query"]
    memory = get_memory_manager()

    # Strategy 1: Semantic search (relevance)
    semantic_results = memory.search_long_term(query, top_k=5)

    # Strategy 2: Recency (recent conversations)
    recent_results = memory.get_episodic_memory(
        agent_name="assistant",
        limit=3
```

```
    )

    # Strategy 3: Exact keyword match
    keyword_results = [
        r for r in semantic_results
        if query.lower() in r["content"].lower()
    ]

    # Merge and rank
    combined = {
        "semantic": semantic_results,
        "recent": recent_results,
        "exact": keyword_results
    }

    # Prioritize: exact match > recent > semantic
    if keyword_results:
        best = keyword_results[0]
    elif recent_results:
        best = recent_results[0]
    else:
        best = semantic_results[0] if semantic_results else None

    return {"best_result": best, "all_results": combined}
```

2.4.5.3 Memory as Cache vs Source of Truth

Memory as cache (can be cleared):

```
# Store expensive computation results
def expensive_analysis(data):
    # Check cache first
    cached = memory.search_long_term(f"analysis:{data}", top_k=1)
    if cached and cached[0]["similarity"] > 0.99:
        return cached[0]["result"]

    # Compute if not cached
    result = perform_expensive_analysis(data)

    # Cache for next time
```

```
    memory.store_long_term(
        content=f"analysis:{data}",
        metadata={"result": result, "cached_at": time.time()}
    )

    return result
```

Memory as source of truth (permanent):

```
# Store user preferences permanently
def store_user_preference(user_id, preference, value):
    memory.store_semantic(
        agent_name="user_prefs",
        fact=f"User {user_id} prefers {preference}={value}",
        metadata={
            "user_id": user_id,
            "preference": preference,
            "value": value,
            "permanent": True
        }
    )
```

2.4.5.4 Debugging Memory Issues

```
from praval.memory import get_memory_manager

def debug_memory_system():
    """Diagnose memory problems."""

    memory = get_memory_manager()

    # Check what's in each layer
    print("=== Short-term Memory ===")
    short_term = memory.get_short_term_memory()
    print(f"Entries: {len(short_term)}")

    print("\n=== Long-term Memory ===")
    # Sample search to see what's stored
    sample = memory.search_long_term("test", top_k=10)
    print(f"Sample results: {len(sample)}")
```

```
print("\n=== Episodic Memory ===")
agents = ["assistant", "researcher", "analyst"]
for agent in agents:
    history = memory.get_episodic_memory(agent, limit=5)
    print(f"{agent}: {len(history)} entries")

print("\n=== Memory Stats ===")
stats = memory.get_stats()
print(f"Total entries: {stats}")
```

Chapter 3

PART III: PRODUCTION

3.1 Chapter 7: Advanced Features

3.1.1 Tool System

Equip agents with external capabilities:

```
from praval.tools import tool

@tool(name="web_search", description="Search the web")
def search_web(query: str) -> str:
    return perform_search(query)

@agent("researcher", tools=[search_web])
def research_agent(spore):
    # Agent can now use web search
    results = search_web(spore.knowledge['query'])
    return {"results": results}
```

3.1.2 Storage Backends

Multiple storage options:

```
from praval.storage import get_storage

# Auto-selects based on configuration
storage = get_storage()

# Save/load data
```

```
storage.save("key", {"data": "value"})
data = storage.load("key")
```

Supported backends:

- **Filesystem**: Local file storage (default)
- **PostgreSQL**: Relational database
- **Redis**: Key-value cache
- **S3**: Object storage
- **Qdrant**: Vector database

Configuration:

```
PRAVAL_STORAGE_BACKEND=postgresql
PRAVAL_DB_URL=postgresql://user:pass@localhost/db
```

3.1.3 Secure Messaging

Enterprise-grade encryption for distributed deployments:

```
from praval.core.secure_reef import SecureReef

reef = SecureReef(
    transport_protocol='amqp',
    amqp_url='amqps://user:pass@rabbitmq:5671/secure',
    tls_config={
        'ca_cert': '/certs/ca.pem',
        'client_cert': '/certs/client.pem',
        'client_key': '/certs/client.key'
    }
)
```

Features:

- End-to-end encryption (Curve25519 + XSalsa20 + Poly1305)
- Digital signatures (Ed25519)
- Multi-protocol support (AMQP, MQTT, STOMP)
- Automatic key rotation

3.2 Chapter 8: Deployment & Best Practices

3.2.1 Docker Deployment

```
# Development
docker-compose up -d

# Production with security
docker-compose -f docker/docker-compose.secure.yml up -d
```

docker-compose.yml:

```
version: '3.8'
services:
  praval:
    build: .
    environment:
      - OPENAI_API_KEY=${OPENAI_API_KEY}
      - PRAVAL_LOG_LEVEL=INFO
    volumes:
      - ./:/app

  qdrant:
    image: qdrant/qdrant
    ports:
      - "6333:6333"
```

3.2.2 Best Practices

3.2.2.1 Agent Design

- **One responsibility per agent**: Keep agents focused
- **Clear identity**: Use descriptive docstrings
- **Idempotent operations**: Safe to retry
- **Meaningful return values**: Structured data

3.2.2.2 Communication Patterns

- **Broadcast for fan-out**: One-to-many distribution
- **Direct messaging for coordination**: One-to-one communication
- **Channels for organization**: Group related agents
- **Type-based routing**: Use `responds_to` effectively

3.2.2.3 Error Handling

```
@agent("processor")
def process_data(spore):
    try:
        result = risky_operation(spore.knowledge["data"])
        return {"result": result}
    except Exception as e:
        # Log error, broadcast failure
        broadcast({"type": "processing_failed", "error": str(e)})
        return {"error": str(e)}
```

3.2.2.4 Performance Optimization

- **Use memory caching**: Reduce redundant LLM calls
- **Enable sampling**: 10% in production
- **Pool expensive resources**: Database connections, HTTP clients
- **Monitor costs**: Track LLM token usage

3.2.2.5 Testing

```
import pytest
from praval import agent, start_agents

def test_research_agent():
    results = []

    @agent("researcher", responds_to=["test"])
    def research_agent(spore):
        result = {"findings": "test findings"}
        results.append(result)
        return result

    start_agents(research_agent,
        initial_data={"type": "test", "query": "test"})

    assert len(results) == 1
    assert "findings" in results[0]
```

3.2.3 Troubleshooting

3.2.3.1 Common Issues

Agent not responding:

- Check `responds_to` matches message type
- Verify agent is registered with `start_agents()`
- Check channel configuration

Memory errors:

- Install memory extras: `pip install praval[memory]`
- Check ChromaDB is accessible
- Verify disk space for storage

Performance issues:

- Enable observability to identify bottlenecks
- Check LLM latency and costs
- Review agent coordination patterns
- Consider caching frequently accessed data

Observability not working:

```
from praval.observability import get_config
config = get_config()
print(f"Enabled: {config.is_enabled()}")
```

3.2.3.2 Debug Mode

```
PRAVAL_LOG_LEVEL=DEBUG python your_script.py
```

3.2.3.3 Getting Help

- GitHub Issues: Bug reports and features
- Documentation: Complete guides in `docs/`
- Examples: Working code in `examples/`
- Discussions: Community Q&A

Chapter 4

APPENDIX

4.1 Quick Reference

4.1.1 Core API

```
from praval import agent, chat, broadcast, start_agents

# Create agent
@agent("name", responds_to=["type1", "type2"])
def my_agent(spore):
    """I do something specific."""
    result = chat(f"Process: {spore.knowledge['data']}")
    return {"result": result}

# Broadcast message
broadcast({"type": "event", "data": value})

# Start agents
start_agents(agent1, agent2, agent3,
    initial_data={"type": "start", "data": initial})
```

4.1.2 Memory

```
from praval.memory import get_memory_manager

memory = get_memory_manager()
```

```
# Store conversation
memory.store_conversation_turn(agent, user_msg, assistant_msg)

# Search
results = memory.search_long_term(query, top_k=5)

# History
history = memory.get_episodic_memory(agent_name)
```

4.1.3 Observability

```
from praval.observability import show_recent_traces, export_traces_to_otlp

# View traces
show_recent_traces(limit=10)

# Export
export_traces_to_otlp("http://localhost:4318/v1/traces")
```

4.1.4 Configuration

```
from praval import configure

configure({
    "default_provider": "openai",
    "default_model": "gpt-4o-mini",
    "max_concurrent_agents": 10,
    "reef_config": {
        "channel_capacity": 1000,
        "message_ttl": 3600
    }
})
```

4.1.5 Environment Variables

```
# LLM Providers
OPENAI_API_KEY=...
ANTHROPIC_API_KEY=...
COHERE_API_KEY=...
```

```
# Framework
PRAVAL_DEFAULT_PROVIDER=openai
PRAVAL_DEFAULT_MODEL=gpt-4o-mini
PRAVAL_LOG_LEVEL=INFO
PRAVAL_MAX_THREADS=10

# Observability
PRAVAL_OBSERVABILITY=auto
PRAVAL_OTLP_ENDPOINT=http://localhost:4318/v1/traces
PRAVAL_SAMPLE_RATE=1.0

# Memory
QDRANT_URL=http://localhost:6333

# Storage
PRAVAL_STORAGE_BACKEND=filesystem
```

4.2 Examples

- **examples/001-009**: Core patterns and capabilities
- **examples/venturelens.py**: Complete business analyzer
- **examples/pythonic_knowledge_graph.py**: Knowledge graph mining
- **examples/rag_chatbot.py**: Memory-enabled chatbot
- **examples/observability/**: Tracing demonstrations

For latest updates: https://github.com/aiexplorations/praval

PyPI Package: https://pypi.org/project/praval/