Praval: A Short Manual

Or: How I Learned to Stop Worrying and Let Simple Agents Build Complex Intelligence

---

Praval (   ) - Sanskrit for coral. Not chosen lightly.

The Problem I Couldn't Stop Thinking About

It was 3 AM on a Tuesday, and I was debugging a monolithic AI agent that had grown, organically and terrifyingly, into something no single person could hold in their head anymore. You know the kind—started simple enough, just a helpful assistant, but then it needed to do research, and then analysis, and then it needed to remember things, and format reports, and somewhere around the fifth "just one more feature" it became a 2,000-line ball of conditionals and state management that failed in mysterious ways.

I remember thinking: *There has to be a better way.*

The irony wasn't lost on me. Here I was, building AI systems by making them do more, know more, be more—when nature had already shown us the opposite pattern works better. Ant colonies don't have genius ants. Bee hives don't have a single bee that knows everything. And coral reefs—those impossibly complex, vibrant ecosystems—they're built by organisms so simple you could describe one in a paragraph.

Complexity emerges from collaboration, not from complexity.

That realization led to Praval. Not as a framework—frameworks are boring. As an exploration of a question: What if we stopped trying to build intelligent agents, and instead built ecosystems where intelligence could emerge?

The Coral Reef Moment

Let me tell you about coral.

A coral polyp is absurdly simple. It's a small creature that does basically one thing: it filters nutrients from water and builds calcium carbonate. That's it. No complex decision-making. No sophisticated reasoning. Just a specialist doing one job well.

But when thousands of these specialists work together, something extraordinary happens. They create structures so complex that they become the foundation for entire ecosystems. Fish that exist nowhere else. Algae that evolved specifically for these conditions. Symbiotic relationships layered upon symbiotic relationships. A reef is more than the sum of its polyps—it's an emergent system where the whole becomes something none of the parts could be alone.

Here's what struck me: **The polyps don't know they're building a reef.** They're just doing their one thing, broadcasting their outputs, consuming what they need from their environment. The intelligence—the system-level intelligence—emerges from the *interaction*, not from any individual component.

Now imagine that with AI agents.

Instead of one massive agent that tries to do everything (and inevitably does some things poorly), what if you had specialists? A researcher who excels at finding information. An analyst who sees patterns. A writer who crafts compelling narratives. A curator who remembers what matters.

Each one simple enough that you can understand it completely. Each one excellent at its specialty. And together, through nothing more than clear communication and well-defined identities, they create intelligence that feels almost magical.

That's Praval.

Why This Matters More Than You Might Think

We're at an interesting moment in AI development. The models themselves—GPT-4, Claude, Gemini—they're increasingly capable. Astonishingly so. But as anyone building with them knows, raw capability isn't the same as useful intelligence.

The real challenge isn't "can an LLM do X?" It's: - How do you build systems that are maintainable? - How do you create AI that gets better over time, not just more complicated? - How do you make something robust enough for production? - How do you build AI you can actually understand?

Traditional approaches give you monoliths. Frameworks give you microservices. Praval gives you something else: **an ecosystem.**

And here's the thing about ecosystems—they scale naturally. You don't "scale" a monolith without eventually hitting the limits of complexity. But ecosystems? They scale by adding specialists. By evolving new relationships. By letting the system discover its own patterns.

This isn't just a different way to organize code. It's a different way to think about intelligence itself.

Your First Agent: Discovery Through Doing

Enough philosophy for now. Let me show you what this feels like in practice.

Here's the moment where abstract concepts become real: your first Praval agent. We're going to create a philosopher—not because philosophy is particularly useful (though it is), but because philosophy is about thinking clearly. And watching an agent think clearly is how you understand what's actually happening.

First, the setup. One line:

```
pip install praval
```

Then create a file. Call it anything. I'll call mine `first_agent.py` because I'm practical that way.

Now here's the entire agent:

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

@agent("philosopher")
def philosophical_agent(spore):
    """I think deeply about questions and explore different perspectives."""

    question = spore.knowledge.get("question")
    response = chat(f"""
    You are a philosopher who thinks carefully about questions.
    Consider this question from multiple angles: {question}

    Provide a thoughtful, multi-perspective analysis.
    """)

    return {"response": response}

# Start the agent with a question
start_agents(
    philosophical_agent,
    initial_data={"question": "What makes a good life?"}
)
```

That's it. That's a complete, functional AI agent.

Let me tell you what just happened, because it matters more than the simplicity suggests.

The Decorator is Not Magic (But It Feels Like It)

That `@agent("philosopher")` line—it transforms an ordinary Python function into something that can: - Register itself in a global agent registry - Receive structured messages (we call them "spores"—more on that metaphor in a moment) - Communicate with other agents - Maintain identity across invocations - Be composed with other agents into larger systems

You didn't write any of that. You just described **what the agent is**: a philosopher.

Identity Over Instructions

Look at that docstring: *"I think deeply about questions and explore different perspectives."*

That's not documentation. That's identity. It tells the agent what it **is**, not what it **does**. This distinction matters enormously.

When you tell an agent what to do ("step 1: analyze input, step 2: generate response…"), you're writing imperative code. It's brittle. It fails when conditions change. It can't adapt.

But when you tell an agent what it **is** ("you're a philosopher who thinks deeply"), you're defining behavior through identity. And identity is robust. A philosopher encountering a question they've never seen before doesn't need new instructions—they think philosophically because that's who they are.

The Spore Protocol

See that `spore.knowledge.get("question")`? That's accessing a message. In Praval, agents communicate through "spores"—structured knowledge packets that carry semantic meaning.

Why "spores"? Because in a coral reef, polyps communicate by releasing chemical signals into the water. They broadcast information that other polyps can absorb and respond to. Our spores work the same way: they carry knowledge through the system, and agents that care about that knowledge respond to it.

A spore is just JSON:

```
{
    "id": "unique_identifier",
    "type": "question",
    "knowledge": {
        "question": "What makes a good life?",
        "context": "philosophical_inquiry"
    },
    "from_agent": "user",
    "timestamp": "2025-10-16T..."
}
```

But like how neurons pass chemical signals that enable thought, spores enable agent cognition.

Run It. Watch What Happens.

```
python first_agent.py
```

The agent receives your question. It thinks (by calling an LLM, but abstracted behind `chat()`). It returns a thoughtful, multi-perspective analysis.

Simple. Clean. Understandable.

Now here's where it gets interesting.

The Moment of Emergence: When Two Become More

One agent is useful. Two agents collaborating is where the magic starts.

Let's add a critic. Not because we want to make our philosopher feel bad, but because good thinking benefits from examination. From questioning. From a second perspective.

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

@agent("philosopher", responds_to=["question"])
def philosophical_agent(spore):
    """I think deeply about questions and explore different perspectives."""

    question = spore.knowledge.get("question")

    response = chat(f"""
    You are a philosopher who thinks carefully about questions.
    Consider this question from multiple angles: {question}

    Provide a thoughtful, multi-perspective analysis.
    """)

    print(f"  Philosopher: {response}\n")
    # Broadcast this thinking to other agents
    broadcast({
        "type": "philosophical_analysis",
        "original_question": question,
        "analysis": response
    })

    return {"response": response}


@agent("critic", responds_to=["philosophical_analysis"])
def critical_agent(spore):
    """I examine ideas for assumptions and weaknesses."""

    analysis = spore.knowledge.get("analysis")
    original_question = spore.knowledge.get("original_question")

    critique = chat(f"""
    You are a critical thinker who examines arguments carefully.

    Original question: {original_question}
    Analysis provided: {analysis}

    What assumptions are being made? What perspectives might be missing?
    What questions does this analysis raise?
    """)

    print(f"  Critic: {critique}\n")

    return {"critique": critique}


# Start both agents
start_agents(
    philosophical_agent,
    critical_agent,
    initial_data={"type": "question", "question": "What makes a good life?"}
)
```

Run this. Watch what happens.

The philosopher receives a question. Thinks. Broadcasts their analysis using `broadcast()`. The critic, configured with `responds_to=["philosophical_analysis"]`, only activates when that specific type of message appears. It examines the philosopher's thinking. Raises questions. Points out assumptions.

Neither agent knows the other exists. They're just doing their jobs. Broadcasting findings. Responding to relevant information.

But together, they create something more valuable than either could alone: a dialectic. A conversation. Thinking that refines itself through interaction.

This is emergence. And it scales.

The Third Agent: When You See The Pattern

Let's add one more agent. A synthesizer.

```python
@agent("synthesizer", responds_to=["philosophical_analysis", "critique"])
def synthesis_agent(spore):
    """I integrate different perspectives into coherent insights."""

    # This agent waits for both the analysis and the critique
    # Then it synthesizes them

    analysis = spore.knowledge.get("analysis", "")
    critique = spore.knowledge.get("critique", "")

    synthesis = chat(f"""
    You integrate different perspectives into coherent understanding.

    Given this analysis and this critique, what deeper insights emerge?
    How do these perspectives complement each other?

    Analysis: {analysis}
    Critique: {critique}
    """)

    print(f"  Synthesis: {synthesis}\n")

    return {"synthesis": synthesis}
```

Now you have three specialists: - A **philosopher** who explores ideas broadly - A **critic** who examines rigorously - A **synthesizer** who integrates perspectives

Run all three together. Watch the conversation unfold.

The philosopher broadcasts analysis. The critic responds with examination. The synthesizer waits for both, then creates something new—insight that emerges from the dialectic.

You didn't orchestrate this. You didn't write a controller that calls agent A, then agent B, then agent C. You just defined what each agent **is**, what they **respond to**, and let the system discover its own flow.

This is how intelligence emerges in Praval. Not from carefully orchestrated complexity, but from simple specialists interacting through clear protocols.

What You Just Discovered

Take a moment to appreciate what we've built in less than 100 lines of code:

1. **Decoupled Intelligence**: Each agent is independent. You can test them separately. Modify one without touching others. Swap in better implementations.
2. **Natural Scaling**: Need more perspectives? Add an ethicist. A pragmatist. A historian. Each is just another specialist.
3. **Emergent Behavior**: The system's intelligence emerges from interactions, not from any single agent's sophistication.
4. **Self-Organization**: Agents discover their own collaboration patterns based on message types and responses.
5. **Clear Communication**: Every interaction is visible, loggable, debuggable. No hidden state. No spooky action at a distance.

This is the coral reef principle in action. Simple organisms. Clear communication protocols. Complex ecosystems emerging.

Memory: When Agents Remember

Here's something that matters more than it might seem at first: continuity.

When you talk to an agent that has no memory, every conversation is like talking to a stranger. Useful, perhaps. But limited. You can't build on shared context. Can't reference past conversations. Can't watch understanding deepen over time.

Memory changes everything.

Praval gives agents multiple kinds of memory—short-term working memory, long-term semantic storage, episodic conversation history. But let me show you the simplest version first, because the simplest version already transforms what's possible:

```python
from praval import agent, chat, start_agents
from praval.memory import MemoryManager, MemoryQuery

# Initialize memory system
memory = MemoryManager(
    qdrant_url="http://localhost:6333",  # Vector database
    collection_name="philosopher_memories"
)

@agent("remembering_philosopher")
def philosophical_agent(spore):
    """I think deeply about questions and remember our conversations."""

    question = spore.knowledge.get("question")
    agent_id = "remembering_philosopher"

    # Search for relevant memories
    relevant_memories = memory.search_memories(MemoryQuery(
        query_text=question,
        agent_id=agent_id,
        limit=3
    ))

    # Get conversation context
    past_context = memory.get_conversation_context(
        agent_id=agent_id,
        turns=5  # Last 5 conversation turns
    )

    # Generate response using memory context
    memory_context = "\n".join([m.content for m in relevant_memories.entries])

    response = chat(f"""
    Based on our previous conversations: {memory_context}

    Recent discussion: {past_context}

    New question: {question}

    Provide a response that builds on our shared understanding.
    """)

    # Store this interaction
    memory.store_conversation_turn(
        agent_id=agent_id,
        user_message=question,
        agent_response=response
    )

    # Store important insights for long-term retention
    memory.store_memory(
        agent_id=agent_id,
        content=f"Discussed: {question} - Key insight: {response[:200]}...",
        importance=0.8  # High importance = longer retention
    )

    return {"response": response}
```

What changed? The agent can now:

1. **Search relevant past conversations** using semantic similarity (that's the vector database magic)
2. **Recall recent context** to maintain conversation continuity
3. **Store new interactions** with importance weighting
4. **Build understanding over time** rather than starting fresh each time

This isn't just about storing data. It's about creating continuity. About building relationships. About agents that get better at helping you because they know you.

Run this agent multiple times with different questions. Watch how later responses reference earlier conversations. How understanding deepens. How the agent develops a model of what matters to you.

It's the difference between talking to ChatGPT and talking to someone who knows you. That difference is everything.

The Architecture of Memory

Praval implements what cognitive scientists call a multi-store memory model:

Short-term Memory (Working Memory): - Fast, in-process storage - Holds ~1000 recent items - Automatic cleanup after 24 hours - Like your own working memory—immediately accessible but temporary

Long-term Memory (Semantic Storage): - Persistent vector database (Qdrant) - Millions of memories - Semantic search using embeddings - Like your long-term memory—slower to access but permanent

Episodic Memory (Conversation History): - Structured timeline of interactions - Maintains conversation coherence - Enables learning from experience - Like remembering not just facts, but the story of how you learned them

Semantic Memory (Knowledge Base): - Facts, concepts, relationships - Domain expertise storage - Knowledge validation and confidence scoring - Like your accumulated understanding of the world

This isn't arbitrary. It mirrors how human cognition actually works. And it works for the same reasons it works in humans: different kinds of memory serve different purposes, and together they create something more powerful than any single storage system could.

Practical Patterns: The Why Behind the How

Let me share patterns I've discovered building real systems with Praval. Not just "here's how to do X," but *why* these patterns work and when to use them.

Pattern 1: The Specialist Pipeline

When: You have a complex process that naturally breaks into stages

Why it works: Each stage can be optimized independently, failures are isolated, you can swap implementations without touching other stages

```python
@agent("extractor", responds_to=["raw_document"])
def extract_content(spore):
    """I extract and clean content from documents."""
    # Specialization: this agent knows document formats deeply
    return {"type": "extracted_content", "text": extracted_text}

@agent("analyzer", responds_to=["extracted_content"])
def analyze_content(spore):
    """I analyze content for key themes and concepts."""
    # Specialization: this agent understands semantic analysis
    return {"type": "analysis_complete", "themes": themes}
```

```python
@agent("reporter", responds_to=["analysis_complete"])
def generate_report(spore):
    """I create well-formatted reports."""
    # Specialization: this agent excels at narrative and formatting
    return {"report": formatted_report}
```

This pattern is powerful because it makes complexity manageable. Each agent has a focused job. You can test them independently. Replace the analyzer with a better one without touching the extractor or reporter.

Pattern 2: The Collaborative Decision

When: You need multiple perspectives before making a decision

Why it works: Different agents can represent different concerns or expertise, decisions benefit from multiple viewpoints

```python
@agent("technical_reviewer", responds_to=["proposal"])
def technical_review(spore):
    """I evaluate technical feasibility."""
    broadcast({"type": "technical_assessment", "assessment": assessment})

@agent("business_reviewer", responds_to=["proposal"])
def business_review(spore):
    """I evaluate business viability."""
    broadcast({"type": "business_assessment", "assessment": assessment})

@agent("decision_maker", responds_to=["technical_assessment", "business_assessment"])
def make_decision(spore):
    """I integrate multiple assessments into decisions."""
    # Wait for both assessments, then decide
    return {"decision": final_decision}
```

This mirrors how good decisions are actually made: by considering multiple perspectives and constraints.

Pattern 3: The Learning System

When: You want agents that improve based on feedback

Why it works: Memory + pattern recognition = learning

```python
@agent("learning_recommender")
def recommender(spore):
    """I recommend solutions based on past successes."""

    query = spore.knowledge.get("problem")

    # Find similar past problems
    similar_cases = memory.search_memories(MemoryQuery(
        query_text=query,
        memory_types=[MemoryType.EPISODIC],
        limit=5
    ))

    # Find what worked before
    successful_patterns = [
        case for case in similar_cases.entries
        if case.metadata.get("success") == True
    ]

    # Generate recommendation based on learned patterns
    recommendation = chat(f"""
    Similar problems in the past: {similar_cases}
    What worked well: {successful_patterns}

    Current problem: {query}

    Recommend an approach based on past successes.
    """)

    return {"recommendation": recommendation}

@agent("feedback_processor", responds_to=["implementation_result"])
def process_feedback(spore):
    """I learn from outcomes."""

    recommendation_id = spore.knowledge.get("recommendation_id")
    success = spore.knowledge.get("success")

    # Update memory with outcome
    memory.store_memory(
        agent_id="learning_recommender",
        content=f"Recommendation {recommendation_id}: {'successful' if success else 'unsuccess
        metadata={"success": success, "recommendation": recommendation_id},
        importance=0.9  # High importance - we learn from outcomes
    )
```

This is where AI gets really interesting: systems that improve not by retraining models, but by accumulating experience and pattern recognition.

Pattern 4: The Resilient System

When: Failures must be contained, not cascading

Why it works: Agent independence means one failure doesn't bring down the system

```
@agent("processor", responds_to=["task"])
def process_with_fallback(spore):
    """I process tasks with graceful fallback."""

    try:
        result = complex_processing(spore.knowledge)
        broadcast({"type": "processing_complete", "result": result})
    except Exception as e:
        # Failure is isolated to this agent
        log_error(e)

        # Broadcast a fallback result
        broadcast({
            "type": "processing_complete",
            "result": simple_fallback(spore.knowledge),
            "fallback_used": True
        })

@agent("monitor", responds_to=["processing_complete"])
def monitor_system_health(spore):
    """I watch for patterns of failure."""

    if spore.knowledge.get("fallback_used"):
        # Track failures, alert if patterns emerge
        track_failure_pattern()
```

Resilience through isolation. Because a coral reef doesn't die if one polyp fails.

The Practice: Building Your Own System

Let me walk you through designing a real system. Not a toy example, but something you might actually build.

The Goal: An intelligent document analysis system that processes research papers, extracts key findings, identifies relationships between papers, and maintains a knowledge base that gets smarter over time.

Traditional Approach: One big agent that does everything. Gets complicated fast. Hard to maintain. Difficult to improve any one aspect without risking others.

Praval Approach: A constellation of specialists.

Let's design it:

The Specialists We Need:

1. **Document Processor**: Understands PDF structure, extracting text cleanly
2. **Content Analyzer**: Identifies key concepts, methods, findings
3. **Relationship Detector**: Finds connections between papers
4. **Knowledge Curator**: Manages the knowledge base, decides what to keep
5. **Query Responder**: Answers questions using accumulated knowledge

The Communication Flow:

```
New Paper → Document Processor → Content Analyzer → Relationship Detector
                                        ↓
                            Knowledge Curator ← Query Responder → User
```

The Implementation:

```python
from praval import agent, chat, broadcast, start_agents
from praval.memory import MemoryManager, MemoryQuery, MemoryType

memory = MemoryManager()

@agent("document_processor", responds_to=["new_paper"])
def process_document(spore):
    """I extract and clean content from academic papers."""

    pdf_path = spore.knowledge.get("pdf_path")

    # Extract text (using whatever PDF library you prefer)
    text = extract_pdf_text(pdf_path)

    # Clean and structure
    structured = {
        "title": extract_title(text),
        "abstract": extract_abstract(text),
        "sections": extract_sections(text),
        "references": extract_references(text)
    }

    broadcast({
        "type": "document_processed",
        "document_id": generate_id(pdf_path),
        "content": structured
    })

@agent("content_analyzer", responds_to=["document_processed"])
def analyze_content(spore):
    """I identify key concepts, methods, and findings."""

    content = spore.knowledge.get("content")
    doc_id = spore.knowledge.get("document_id")

    analysis = chat(f"""
    Analyze this research paper:
    Title: {content['title']}
    Abstract: {content['abstract']}
    Extract:
    - Key concepts and terminology
    - Research methods used
    - Main findings
    - Limitations noted

    Format as structured JSON.
    """)

    parsed_analysis = parse_json(analysis)

    broadcast({
        "type": "content_analyzed",
        "document_id": doc_id,
        "analysis": parsed_analysis
    })

@agent("relationship_detector", responds_to=["content_analyzed"])
def detect_relationships(spore):
    """I find connections between papers."""

    doc_id = spore.knowledge.get("document_id")
    analysis = spore.knowledge.get("analysis")

    # Search for related papers in memory
    related_papers = memory.search_memories(MemoryQuery(
        query_text=f"{analysis['concepts']} {analysis['methods']}",
        memory_types=[MemoryType.SEMANTIC],
        limit=10
    ))

    if related_papers.entries:
        relationships = chat(f"""
        This paper: {analysis}
        Related papers: {related_papers}

        Identify:
        - Which papers cite similar work
        - Which use similar methods
        - Which have conflicting findings
        - Which extend or build upon each other
        """)

        broadcast({
            "type": "relationships_found",
            "document_id": doc_id,
            "relationships": relationships
        })
```

```
@agent("knowledge_curator", responds_to=["content_analyzed", "relationships_found"])
def curate_knowledge(spore):
    """I maintain the knowledge base, deciding what to store and how."""

    doc_id = spore.knowledge.get("document_id")
    analysis = spore.knowledge.get("analysis", {})
    relationships = spore.knowledge.get("relationships", "")

    # Determine importance based on novelty and connection strength
    importance = calculate_importance(analysis, relationships)

    # Store in long-term semantic memory
    memory.store_knowledge(
        agent_id="knowledge_base",
        knowledge=f"Paper: {analysis.get('title', '')} - Concepts: {analysis.get('concepts', '
        domain="research_papers",
        confidence=0.9,
        metadata={
            "document_id": doc_id,
            "relationships": relationships
        }
    )

    print(f"  Added to knowledge base with importance: {importance}")

@agent("query_responder", responds_to=["research_query"])
def respond_to_queries(spore):
    """I answer questions using accumulated knowledge."""

    query = spore.knowledge.get("query")

    # Search the knowledge base
    relevant_knowledge = memory.search_memories(MemoryQuery(
        query_text=query,
        agent_id="knowledge_base",
        memory_types=[MemoryType.SEMANTIC],
        limit=5
    ))

    # Generate comprehensive answer
    answer = chat(f"""
    Question: {query}

    Relevant research from knowledge base:
    {format_memories(relevant_knowledge)}

    Provide a comprehensive answer citing specific papers.
    """)

    return {"answer": answer}
```

What We Just Built:

A system that: - **Processes documents** without your papers architecture intervention - **Extracts structured knowledge** using LLM intelligence - **Discovers relationships** between papers automatically - **Maintains a growing knowledge base** that gets richer over time - **Answers questions** with citation-backed responses

And crucially: each piece is understandable, testable, and replaceable.

Want a better relationship detector? Swap it in. Need to process different document types? Replace just the document processor. Want to add a new agent that summarizes recent additions? Just add it—no need to touch existing agents.

This is the power of the specialist pattern at scale.

Closing Thoughts: What We're Really Building

It's late now, and I'm back at my desk thinking about what Praval actually represents.

On one level, it's a framework. A way to organize AI agents. Decorators and message passing and memory systems. All very technical and practical.

But I think it's something more.

We're at this interesting moment where AI capabilities are advancing faster than our ability to use them well. We have models that can write, reason, analyze, create—but building systems with them often feels like fighting against their nature. Like trying to make a river flow uphill.

Praval is an attempt to work with the grain instead of against it.

The coral reef metaphor isn't just cute. It's profound. Because what makes a reef work isn't central planning or perfect coordination. It's simple organisms doing what they're good at, communicating clearly, and letting complex behavior emerge.

When you build with Praval, you're not just organizing code differently. You're adopting a different epistemology—a different way of thinking about what intelligence is and how it should be structured.

Intelligence isn't a monolith. It's not something you cram into ever-larger models until they can "do everything." Intelligence is specialized capabilities collaborating effectively. It's the conversation between different ways of thinking. It's emergence from interaction.

That's true for human intelligence too, by the way. Your brain isn't one generalized thinking machine. It's specialized regions communicating, integrating perspectives, creating the experience of unified thought from distributed processing. The prefrontal cortex doesn't try to be the visual cortex. They specialize and collaborate.

Maybe the future of AI looks less like AGI—Artificial General Intelligence—and more like AEI: Artificial Emergent Intelligence. Systems where capabilities arise from the right specialists working together in the right ways.

That's what excites me about this work. Not that we're building better chatbots or more efficient agents. But that we're exploring a fundamentally different architecture for intelligence itself.

What's Next: Where to Go From Here

If you've made it this far, you understand the core ideas. You've seen simple agents become collaborative systems. You've watched emergence happen in real code.

Here's what to explore next:

The Complete Manual

For deeper technical details, architectural patterns, and advanced features, see the complete Praval manual. It covers: - Detailed architecture specifications - Advanced memory system features - Production deployment patterns - Security and scalability - The full tool system - Distributed agent architectures

The Examples

The `examples/` directory contains real, working systems: - `001_single_agent_identity.py` - Understanding agent identity - `003_specialist_collaboration.py` - Multi-agent patterns - `005_memory_enabled_agents.py` - Agents that remember and learn - `009_emergent_collective_intelligence.` - Complex emergence in action

Run them. Modify them. Break them and fix them. That's how you really learn.

Your Own System

The best way to understand Praval is to build something that matters to you. Start small: - What task could benefit from multiple perspectives? - What process could be broken into clear stages? - What system would benefit from memory and learning?

Start with two agents. Then add a third. Watch what emerges.

The Community

Praval is open source and evolving. The patterns we're discovering—the ways agents can collaborate, the architectures that work in production, the lessons learned from building real systems—they're being developed by people building real things.

Your discoveries matter. Your patterns matter. Your "oh, this is what works" moments matter.

A Final Word

It's well past 3 AM now. Different Tuesday. Different debugging session.

But this time I'm not fighting a monolith. I'm watching three specialist agents collaborate on a problem, each doing what it does best, and I can see exactly what each is thinking, what decisions they're making, how they're building on each other's work.

When one of them finds an unexpected connection—an insight that emerges from how the analyst's pattern recognition combines with the researcher's knowledge—I don't see it as "the AI did something clever."

I see it as what happens when you build systems that think like coral reefs: simple, clear, collaborative, emergent.

That's what Praval makes possible.

Not just better code. Better thinking.

For technical support, updates, and community discussion: - **GitHub**: github.com/aiexplorations/praval - **Documentation**: `docs/` directory - **Examples**: `examples/` directory

Build something interesting. Then tell us about it.