AI AGENT SYSTEM

Development Roadmap

Phase 1.5 → Phase 5: Complete Implementation Path

Truth Verification Framework

Last Updated: February 2026

ROADMAP OVERVIEW

This roadmap transforms the Phase 1.5 template into a production truth verification system. Each phase builds incrementally, ensuring working functionality at every step.

Phase	**Goal**	**Duration**	**Output**
1.5	Template Foundation	COMPLETE	Production-ready template
2.0	Basic Agent	2-4 weeks	Working PineScript expert
3.0	RAG Integration	4-6 weeks	Documentation-grounded answers
4.0	Verification Layer	4-6 weeks	Hallucination detection
5.0	Production Polish	2-4 weeks	Complete truth system

Total Timeline: 3-5 months to production-ready truth verification system

PHASE 1.5: TEMPLATE FOUNDATION

STATUS: ✅ COMPLETE

What Was Built

- Immutable template with 14 core files
- 7-layer architecture (Agent, Controller, Tool, Memory, Config, Interface, Script)
- Project generator for isolated domain experts
- PostgreSQL with pgvector for vector storage
- Git version control initialized
- Complete documentation

Verification Checklist

☑ All 14 template files present and verified

☑ create_project.py generates working projects

☑ Layer boundaries clearly defined

☑ .gitignore prevents sensitive file commits

☑ Git repository initialized with v1.5 tag

Current State

Template is production-ready but controllers contain only placeholder methods. No actual AI functionality implemented yet.

PHASE 2.0: BASIC AGENT

GOAL: Working PineScript Expert (No RAG)

Build your first functional agent that can answer PineScript questions using only LLM knowledge. No documentation retrieval yet - just prove the architecture works.

Milestones

M2.1: Create PineScript Project (Day 1)

```
cd ~/agents
python3 create_project.py pinescript 54320
cd ~/agents/projects/pinescript-expert

# Add OpenAI API key to .env
nano .env
# Add: OPENAI_API_KEY=sk-your-actual-key

# Start database
docker-compose up -d

# Install dependencies
pip install -r requirements.txt

# Initialize database
python3 scripts/init_db.py
```

Success Criteria:

✓ Project created at ~/agents/projects/pinescript-expert/

✓ Database running on port 54320

✓ No errors during initialization

M2.2: Implement Basic Agent (Days 2-3)

File: agents/pinescript_agent.py

```
from agents.base import BaseAgent

class PineScriptAgent(BaseAgent):
    def __init__(self, config=None):
        system_prompt = """You are an expert in Pine Script, the programming language
used in TradingView.

Your expertise includes:
- Pine Script v6 syntax and functions
- Indicator creation and plotting
- Strategy development and backtesting
- Alert conditions and notifications
- Chart overlays and studies

Guidelines:
1. Provide accurate, working code examples
```

```
2. Explain syntax clearly
3. Reference Pine Script v6 documentation
4. Warn about deprecated features
5. If uncertain, state "I need to verify this in the documentation"

Always include code examples when relevant.
Never make up function names or syntax."""

        super().__init__(
            name='pinescript_expert',
            system_prompt=system_prompt,
            config=config
        )
```

Success Criteria:

✓ File created in agents/ directory

✓ Inherits from BaseAgent correctly

✓ System prompt is domain-specific

M2.3: Implement Basic Controller (Days 4-5)

File: controllers/pinescript_controller.py

```
from controllers.controller import Controller

class PineScriptController(Controller):
    async def _think(self, query, context, tool_results):
        prompt = f"""User Question: {query}

Provide a clear, accurate answer about Pine Script.
If you provide code, make it executable and correct.
If uncertain about syntax, state that explicitly."""

        response = await self.agent.run(prompt)
        return response

    async def _load_context(self, session_id, query):
        return {
            'session_id': session_id,
            'previous_queries': []
        }

    async def _decide_plan(self, query, context):
        return {
            'action': 'direct_response',
            'tools': []
        }

    async def _execute_tools(self, plan):
        return {}

    async def _remember(self, session_id, query, response):
        if session_id:
            await self.memory.store_conversation(
                session_id, query, response
            )
```

Success Criteria:

✓ _think() method implemented

✓ Other methods return valid (empty) data

✓ No errors when called

M2.4: Update Streamlit Interface (Day 6)

File: interfaces/streamlit_app.py

Update the controller initialization section:

```
from agents.pinescript_agent import PineScriptAgent
from controllers.pinescript_controller import PineScriptController

if 'controller' not in st.session_state:
    agent = PineScriptAgent(config=config['agent'])
    memory = Memory(
        namespace=config['memory']['namespace'],
        config=config['memory']
    )
    tools = {
        'db': DatabaseTool(),
        'web': WebTool()
    }
    st.session_state.controller = PineScriptController(
        agent, memory, tools, config
    )
```

Success Criteria:

✓ Imports updated

✓ Controller uses PineScript classes

✓ Streamlit launches without errors

M2.5: Test and Validate (Day 7)

Launch the interface:

```
streamlit run interfaces/streamlit_app.py
```

Test Queries:

1. "How do I plot a simple moving average in Pine Script?"

2. "What is the difference between plot() and plotshape()?"

3. "Show me a basic strategy template"

4. "How do I create alerts in Pine Script v6?"

5. "What does the security() function do?"

Success Criteria:

✓ All queries return relevant answers

✓ Code examples are syntactically correct

✓ No hallucinated function names

✓ Agent admits uncertainty when appropriate

Phase 2.0 Completion Checklist

☐ PineScript project created and running

☐ PineScriptAgent implemented with domain expertise

☐ PineScriptController implements basic orchestration

☐ Streamlit interface updated and functional

☐ 5 test queries answered successfully

☐ Git commit: "Phase 2.0 Complete - Basic PineScript Agent"

☐ Git tag: v2.0

What You Have After Phase 2

- A working PineScript expert agent
- Proof that your architecture works
- Understanding of agent → controller → interface flow
- Foundation for RAG integration

Known Limitations:

- Answers based on LLM training data only
- No access to official documentation
- Cannot verify claims against sources
- May hallucinate deprecated syntax

PHASE 3.0: RAG INTEGRATION

GOAL: Documentation-Grounded Answers

Add Retrieval-Augmented Generation so answers are grounded in official PineScript documentation. This is your truth foundation.

Milestones

M3.1: Documentation Crawler (Week 1)

File: tools/crawler.py

```
import requests
from bs4 import BeautifulSoup
from urllib.parse import urljoin, urlparse
import time

class DocumentationCrawler:
    def __init__(self, base_url, max_pages=100):
        self.base_url = base_url
        self.max_pages = max_pages
        self.visited = set()
        self.session = requests.Session()
        self.session.headers.update({
            'User-Agent': 'Mozilla/5.0 (Documentation Crawler)'
        })

    def crawl(self):
        pages = []
        to_visit = [self.base_url]

        while to_visit and len(pages) < self.max_pages:
            url = to_visit.pop(0)

            if url in self.visited:
                continue

            try:
                response = self.session.get(url, timeout=10)
                response.raise_for_status()
                self.visited.add(url)

                soup = BeautifulSoup(response.text, 'html.parser')

                title = soup.find('title')
                title_text = title.get_text() if title else url

                content = soup.get_text(separator='\n', strip=True)

                pages.append({
                    'url': url,
                    'title': title_text,
                    'content': content
                })

                for link in soup.find_all('a', href=True):
                    next_url = urljoin(url, link['href'])
                    if self._is_valid_url(next_url):
                        to_visit.append(next_url)

                time.sleep(1)

            except Exception as e:
                print(f"Error crawling {url}: {e}")
                continue

        return pages

    def _is_valid_url(self, url):
        parsed = urlparse(url)
        base_parsed = urlparse(self.base_url)
        return (
            parsed.netloc == base_parsed.netloc and
            url not in self.visited and
            not url.endswith(('.pdf', '.zip', '.jpg', '.png'))
        )
```

Success Criteria:

✓ Can crawl TradingView Pine Script docs

✓ Extracts title and content

✓ Respects rate limits (1 second delay)

✓ Returns list of page dictionaries

M3.2: Text Chunking and Embedding (Week 2)

File: tools/embedder.py

```
from openai import OpenAI
import os

class TextEmbedder:
    def __init__(self):
        self.client = OpenAI(api_key=os.getenv('OPENAI_API_KEY'))
        self.model = 'text-embedding-3-small'
        self.chunk_size = 1000
        self.chunk_overlap = 200

    def chunk_text(self, text, metadata=None):
        chunks = []
        words = text.split()

        for i in range(0, len(words), self.chunk_size - self.chunk_overlap):
            chunk_words = words[i:i + self.chunk_size]
            chunk_text = ' '.join(chunk_words)

            chunks.append({
                'text': chunk_text,
                'metadata': metadata or {},
                'position': i
            })

        return chunks
    def embed_text(self, text):
        response = self.client.embeddings.create(
            input=text,
            model=self.model
        )
        return response.data[0].embedding

    def embed_chunks(self, chunks):
        embedded_chunks = []

        for chunk in chunks:
            embedding = self.embed_text(chunk['text'])
            embedded_chunks.append({
                'text': chunk['text'],
                'embedding': embedding,
                'metadata': chunk['metadata'],
                'position': chunk['position']
            })

        return embedded_chunks
```

Success Criteria:

✓ Chunks text into ~1000 word segments

✓ 200 word overlap between chunks

✓ Generates embeddings using OpenAI

✓ Returns embedded chunks with metadata

M3.3: Vector Storage (Week 2)

Update: tools/database.py (add methods)

```
def store_embedding(self, url, title, content, embedding):
    query = """
        INSERT INTO documentation (url, title, content, embedding)
        VALUES (%s, %s, %s, %s)
        ON CONFLICT (url) DO UPDATE
        SET title = EXCLUDED.title,
            content = EXCLUDED.content,
            embedding = EXCLUDED.embedding
    """
    self.execute(query, (url, title, content, str(embedding)))

def semantic_search(self, query_embedding, limit=5):
    query = """
        SELECT url, title, content,
               1 - (embedding <=> %s::vector) AS similarity
        FROM documentation
        ORDER BY embedding <=> %s::vector
        LIMIT %s
    """
    return self.query(query, (str(query_embedding), str(query_embedding), limit))
```

Success Criteria:

✓ store_embedding() inserts docs with vectors

✓ semantic_search() returns top K similar docs

✓ Uses pgvector cosine similarity (<=>)

M3.4: Documentation Ingestion Script (Week 3)

File: scripts/ingest_docs.py

```
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from tools.crawler import DocumentationCrawler
from tools.embedder import TextEmbedder
from tools.database import DatabaseTool

def ingest_documentation():
    base_url = 'https://www.tradingview.com/pine-script-docs/en/v6/'

    print("Crawling documentation...")
    crawler = DocumentationCrawler(base_url, max_pages=50)
    pages = crawler.crawl()
    print(f"Crawled {len(pages)} pages")

    print("Chunking and embedding...")
    embedder = TextEmbedder()
    db = DatabaseTool()

    for page in pages:
        print(f"Processing: {page['title']}")

        chunks = embedder.chunk_text(
            page['content'],
            metadata={'url': page['url'], 'title': page['title']}
        )

        embedded_chunks = embedder.embed_chunks(chunks)

        for chunk in embedded_chunks:
            db.store_embedding(
                url=page['url'],
                title=page['title'],
                content=chunk['text'],
                embedding=chunk['embedding']
            )

    db.close()
    print("Documentation ingestion complete!")

if __name__ == '__main__':
    ingest_documentation()
```

Success Criteria:

✓ Crawls Pine Script documentation

✓ Chunks and embeds all pages

✓ Stores in documentation table

✓ Completes without errors

M3.5: Update Controller for RAG (Week 4)

Update: controllers/pinescript_controller.py

```
from tools.embedder import TextEmbedder

class PineScriptController(Controller):
    def __init__(self, agent, memory, tools, config):
        super().__init__(agent, memory, tools, config)
        self.embedder = TextEmbedder()

    async def _load_context(self, session_id, query):
        query_embedding = self.embedder.embed_text(query)

        relevant_docs = self.tools['db'].semantic_search(
            query_embedding, limit=3
        )

        context = {
            'session_id': session_id,
            'relevant_docs': relevant_docs
        }

        return context

    async def _think(self, query, context, tool_results):
        docs_context = ""
        if context.get('relevant_docs'):
            docs_context = "\n\nRelevant Documentation:\n"
            for doc in context['relevant_docs']:
                docs_context += f"\nSource: {doc['title']}\n{doc['content']}\n"

        prompt = f"""You are a Pine Script expert. Answer based on the official
documentation provided.

User Question: {query}
{docs_context}

Guidelines:
- Base your answer on the documentation provided above
- If the documentation doesn't cover the question, say so
- Include citations to source URLs
- Provide accurate code examples from the docs"""

        response = await self.agent.run(prompt)
        return response
```

Success Criteria:

✓ _load_context() retrieves relevant docs

✓ _think() includes docs in prompt

✓ Answers cite documentation sources

M3.6: Test RAG System (Week 4)

Run ingestion:

```
python3 scripts/ingest_docs.py
```

Test queries (same as Phase 2):

1. "How do I plot a simple moving average in Pine Script?"

2. "What is the difference between plot() and plotshape()?"

3. "Show me a basic strategy template"

Compare Results:

• Phase 2: LLM knowledge only

• Phase 3: Documentation-grounded answers

• Phase 3 should cite specific docs and use current syntax

Success Criteria:

✓ Answers reference official documentation

✓ Citations include source URLs

✓ Code examples match current Pine Script v6

✓ No hallucinated function names

Phase 3.0 Completion Checklist

☐ DocumentationCrawler implemented

☐ TextEmbedder chunks and embeds content

☐ DatabaseTool has vector search methods

☐ Ingestion script successfully loads docs

☐ Controller retrieves relevant documentation

☐ Answers grounded in official sources

☐ Citations included in responses

☐ Git commit: "Phase 3.0 Complete - RAG Integration"

☐ Git tag: v3.0

What You Have After Phase 3

• Truth-grounded responses from official docs

• Semantic search across Pine Script documentation

• Citation tracking for verification

• 40% reduction in hallucinations (estimated)

• Answers reflect current syntax, not outdated LLM training

PHASE 4.0: VERIFICATION LAYER

GOAL: Hallucination Detection and Mitigation

Add a verification agent that checks claims against sources and flags unsupported statements. This is your truth verification core.

Milestones

M4.1: Verification Agent (Week 1)

File: agents/verification_agent.py

```
from agents.base import BaseAgent

class VerificationAgent(BaseAgent):
    def __init__(self, config=None):
        system_prompt = """You are a fact-checking agent. Your job is to verify claims
against source documentation.

For each claim:
1. Check if it's supported by the provided documentation
2. Rate confidence: SUPPORTED, PARTIALLY_SUPPORTED, UNSUPPORTED
3. Explain your reasoning

Output Format:
CLAIM: [the claim being checked]
VERDICT: [SUPPORTED/PARTIALLY_SUPPORTED/UNSUPPORTED]
REASONING: [why you reached this verdict]
SOURCE: [which documentation supports/contradicts this]

Be strict. If documentation doesn't explicitly support a claim, mark it
UNSUPPORTED."""

        super().__init__(
            name='verification_agent',
            system_prompt=system_prompt,
            config=config
        )
```

Success Criteria:

✓ Agent checks claims against sources

✓ Returns structured verdicts

✓ Conservative (prefers UNSUPPORTED when uncertain)

M4.2: Claim Extraction (Week 1-2)

File: tools/claim_extractor.py

```
class ClaimExtractor:
    def extract_claims(self, text):
        claims = []
        sentences = text.split('.')

        for sentence in sentences:
            sentence = sentence.strip()
            if not sentence:
                continue

            if self._is_factual_claim(sentence):
                claims.append({
                    'text': sentence,
                    'type': self._classify_claim(sentence)
                })

        return claims

    def _is_factual_claim(self, sentence):
        non_factual_indicators = [
            'i think', 'i believe', 'maybe', 'perhaps',
            'could be', 'might be', 'possibly'
        ]

        sentence_lower = sentence.lower()
        return not any(ind in sentence_lower for ind in non_factual_indicators)

    def _classify_claim(self, sentence):
        if 'function' in sentence.lower() or '()' in sentence:
            return 'FUNCTION_CLAIM'
        elif 'syntax' in sentence.lower():
            return 'SYNTAX_CLAIM'
        elif 'example' in sentence.lower() or 'code' in sentence.lower():
            return 'EXAMPLE_CLAIM'
        else:
            return 'GENERAL_CLAIM'
```

Success Criteria:

✓ Extracts factual claims from responses

✓ Filters out opinions and uncertainties

✓ Classifies claim types

M4.3: Update Controller for Verification (Week 2-3)

Update: controllers/pinescript_controller.py

```
from agents.verification_agent import VerificationAgent
from tools.claim_extractor import ClaimExtractor

class PineScriptController(Controller):
    def __init__(self, agent, memory, tools, config):
        super().__init__(agent, memory, tools, config)
        self.embedder = TextEmbedder()
        self.verifier = VerificationAgent(config=config.get('agent'))
        self.claim_extractor = ClaimExtractor()

    async def process_query(self, query, session_id=None):
        context = await self._load_context(session_id, query)
        plan = await self._decide_plan(query, context)
        tool_results = await self._execute_tools(plan)
        response = await self._think(query, context, tool_results)

        verified_response = await self._verify_response(
            response, context
        )

        await self._remember(session_id, query, verified_response)
        return verified_response

    async def _verify_response(self, response, context):
        claims = self.claim_extractor.extract_claims(response)

        if not claims:
            return response

        docs_text = "\n\n".join([
            f"{doc['title']}:\n{doc['content']}"
            for doc in context.get('relevant_docs', [])
        ])

        verification_results = []
        unsupported_claims = []

        for claim in claims:
            verification_prompt = f"""Documentation:
{docs_text}

Claim to verify: {claim['text']}

Verify this claim against the documentation above."""

            verdict = await self.verifier.run(verification_prompt)
            verification_results.append(verdict)

            if 'UNSUPPORTED' in verdict:
                unsupported_claims.append(claim['text'])

        if unsupported_claims:
            warning = "\n\n⚠️ WARNING: Some claims could not be verified:\n"
            warning += "\n".join(f"- {claim}" for claim in unsupported_claims)
            warning += "\n\nPlease verify these statements independently."
            return response + warning

        return response + "\n\n✓ All claims verified against documentation"
```

Success Criteria:

✓ Extracts claims from responses

✓ Verifies each claim against sources

✓ Flags unsupported statements

✓ Adds verification status to response

M4.4: Confidence Scoring (Week 3-4)

File: tools/confidence_scorer.py

```
class ConfidenceScorer:
    def score_response(self, response, verification_results):
        total_claims = len(verification_results)

        if total_claims == 0:
            return {
                'score': 0.5,
                'level': 'MEDIUM',
                'reason': 'No factual claims to verify'
            }

        supported = sum(
            1 for v in verification_results
            if 'SUPPORTED' in v and 'UNSUPPORTED' not in v
        )
        partially = sum(
            1 for v in verification_results
            if 'PARTIALLY_SUPPORTED' in v
        )

        score = (supported + 0.5 * partially) / total_claims

        if score >= 0.9:
            level = 'HIGH'
        elif score >= 0.7:
            level = 'MEDIUM'
        else:
            level = 'LOW'

        return {
            'score': score,
            'level': level,
            'supported': supported,
            'partially': partially,
            'total': total_claims
        }
```
Success Criteria:

✓ Calculates confidence based on verifications

✓ Returns HIGH/MEDIUM/LOW rating

✓ Provides explanation of score

M4.5: Test Verification System (Week 4)

Test queries designed to trigger verification:

1. "The strategy() function was deprecated in v6" (FALSE - should flag)

2. "You can use ta.sma() for simple moving average" (TRUE - should verify)

3. "Pine Script supports Python syntax" (FALSE - should flag)

Success Criteria:

✓ True claims marked as verified

✓ False claims flagged as unsupported

✓ Confidence scores reflect verification status

✓ Warnings added to uncertain responses

Phase 4.0 Completion Checklist

☐ VerificationAgent implemented

☐ ClaimExtractor extracts factual claims

☐ Controller verifies claims against sources

☐ ConfidenceScorer rates response accuracy

☐ Unsupported claims flagged with warnings

☐ False statements successfully detected

☐ Git commit: "Phase 4.0 Complete - Verification Layer"

☐ Git tag: v4.0

What You Have After Phase 4

• Automatic hallucination detection

• Claim-by-claim verification against sources

• Confidence scoring for responses

• Warnings for unverified statements

• 60-80% reduction in undetected errors (estimated)

• True "truth verification" system

PHASE 5.0: PRODUCTION POLISH

GOAL: Production-Ready System

Add logging, error handling, monitoring, and UX improvements to make the system production-ready for personal use.

Milestones

M5.1: Logging System (Week 1)

File: tools/logger.py

```
import logging
import json
from datetime import datetime
import os

class StructuredLogger:
    def __init__(self, config):
        self.config = config
        log_dir = config.get('logging', {}).get('output', 'logs/')
        os.makedirs(log_dir, exist_ok=True)

        self.logger = logging.getLogger('agent_system')
        self.logger.setLevel(logging.INFO)

        handler = logging.FileHandler(f'{log_dir}/agent.log')
        handler.setFormatter(logging.Formatter('%(message)s'))
        self.logger.addHandler(handler)

    def log_query(self, query, session_id, context):
        self._log({
            'event': 'query_received',
            'session_id': session_id,
            'query': query,
            'context_size': len(context.get('relevant_docs', []))
        })

    def log_response(self, query, response, confidence, duration):
        self._log({
            'event': 'response_generated',
            'query': query,
            'response_length': len(response),
            'confidence': confidence,
            'duration_ms': duration
        })

    def log_verification(self, claims, results):
        self._log({
            'event': 'verification_complete',
            'total_claims': len(claims),
            'verified': sum(1 for r in results if 'SUPPORTED' in r)
        })

    def _log(self, data):
        data['timestamp'] = datetime.now().isoformat()
        self.logger.info(json.dumps(data))
```

Success Criteria:

✓ Logs all queries and responses

✓ JSON format for easy parsing

✓ Tracks confidence and verification

M5.2: Error Handling (Week 1-2)

Update: controllers/pinescript_controller.py (add error handling)

```
async def process_query(self, query, session_id=None):
    try:
        start_time = time.time()

        context = await self._load_context(session_id, query)
        plan = await self._decide_plan(query, context)
        tool_results = await self._execute_tools(plan)
        response = await self._think(query, context, tool_results)
        verified_response = await self._verify_response(response, context)

        duration = (time.time() - start_time) * 1000

        if hasattr(self, 'logger'):
            self.logger.log_response(
                query, verified_response,
                self._get_confidence(verified_response),
                duration
            )

        await self._remember(session_id, query, verified_response)
        return verified_response

    except Exception as e:
        error_msg = f"Error processing query: {str(e)}"
        if hasattr(self, 'logger'):
            self.logger.log_error(query, str(e))
        return f"I encountered an error: {error_msg}\n\nPlease try rephrasing your
question."
```

Success Criteria:

✓ Graceful error handling

✓ Errors logged for debugging

✓ User-friendly error messages

M5.3: UI Improvements (Week 2)

Update: interfaces/streamlit_app.py (add features)

```
st.title(f"{config['ui']['page_icon']} {config['project']['name']}")

with st.sidebar:
    st.header("Settings")

    show_sources = st.checkbox("Show Sources", value=True)
    show_confidence = st.checkbox("Show Confidence", value=True)
    show_verification = st.checkbox("Show Verification", value=True)

    st.divider()

    if st.button("Clear Conversation"):
        st.session_state.messages = []
        st.rerun()

    st.divider()
    st.caption(f"Version {config['project']['version']}")

if prompt := st.chat_input('Ask me anything...'):
    st.session_state.messages.append({'role': 'user', 'content': prompt})
    with st.chat_message('user'):
        st.markdown(prompt)

    with st.chat_message('assistant'):
        with st.spinner('Thinking...'):
            response = run_async(
                st.session_state.controller.process_query(prompt)
            )

            st.markdown(response)

            if show_confidence and '✓' in response:
                st.success("High confidence - all claims verified")
            elif show_confidence and '⚠️' in response:
                st.warning("Some claims unverified")

            st.session_state.messages.append(
                {'role': 'assistant', 'content': response}
            )
```

Success Criteria:

✓ Sidebar with settings

✓ Toggle sources/confidence display

✓ Clear conversation button

✓ Visual confidence indicators

M5.4: Performance Monitoring (Week 3)

File: tools/metrics.py

```
import time
from collections import defaultdict

class MetricsCollector:
    def __init__(self):
        self.metrics = defaultdict(list)

    def record_latency(self, operation, duration_ms):
        self.metrics[f'{operation}_latency'].append(duration_ms)

    def record_token_usage(self, tokens):
        self.metrics['tokens_used'].append(tokens)

    def get_stats(self):
        stats = {}

        for key, values in self.metrics.items():
            if values:
                stats[key] = {
                    'count': len(values),
                    'avg': sum(values) / len(values),
                    'min': min(values),
                    'max': max(values)
                }

        return stats
```

Success Criteria:

✓ Tracks response latency

✓ Monitors token usage

✓ Calculates performance statistics

M5.5: Documentation Updates (Week 4)

Update project README with:

• Complete setup instructions

• Example queries and expected outputs

• Troubleshooting guide

• Performance benchmarks

• Known limitations

Success Criteria:

✓ README has all usage information

✓ Examples demonstrate all features

✓ Clear documentation of limitations

Phase 5.0 Completion Checklist

☐ Structured logging implemented

☐ Comprehensive error handling

☐ UI improvements (sidebar, settings)

☐ Performance monitoring active

☐ Documentation complete and updated

☐ All features tested end-to-end

☐ Git commit: "Phase 5.0 Complete - Production Polish"

☐ Git tag: v6.0

What You Have After Phase 5

• Production-ready truth verification system

• Professional logging and monitoring

• Robust error handling

• Polished user interface

• Complete documentation

• Personal PineScript expert you can trust

BEYOND PHASE 5: FUTURE ENHANCEMENTS

Phase 6: Multi-Domain Expansion (Optional)

Create additional domain experts using the same template:

• Python expert (port: 54321)

• Shopify expert (port: 54322)

• React expert (port: 54323)

Each expert follows the same Phase 2-5 path:

1. Create project from template

2. Implement domain agent

3. Crawl domain documentation

4. Add verification layer

5. Polish and deploy

Phase 7: Multi-Agent Collaboration (Advanced)

Enable agents to work together:

• PineScript agent generates code

• Python agent validates logic

• Combined expert system

Requires:

• Agent communication protocol

• Task delegation system

• Consensus mechanism

Phase 8: Real-Time Updates (Advanced)

Keep documentation current:

• Scheduled documentation refreshes

• Changelog tracking

- Deprecation warnings
- Version-aware responses

SUCCESS METRICS

Phase 2 Success Criteria

Metric	Target
Query Response Time	< 5 seconds
Code Examples Syntactically Correct	> 90%
Agent Admits Uncertainty	When appropriate
No Runtime Errors	100%

Phase 3 Success Criteria

Metric	Target
Documentation Pages Indexed	> 50
Semantic Search Relevance	> 80%
Answers Include Citations	100%
Hallucination Reduction	~40%

Phase 4 Success Criteria

Metric	Target
False Claims Detected	> 90%
Verification Latency	< 3 seconds
Confidence Accuracy	> 85%
Total Hallucination Reduction	60-80%

Phase 5 Success Criteria

Metric	Target
Error Rate	< 0.1%
User Experience Rating	Excellent
Documentation Completeness	100%
Personal Usage Value	Daily use for 2+ years

TROUBLESHOOTING GUIDE

Common Issues and Solutions

Issue: Crawler Not Finding Documentation

Solution:

• Check base_url is correct

• Verify website is accessible

• Increase max_pages limit

• Check robots.txt compliance

Issue: Embeddings Taking Too Long

Solution:

• Reduce chunk size

• Process in batches

• Use faster embedding model

• Add progress indicators

Issue: Semantic Search Returns Irrelevant Results

Solution:

• Increase number of results (top K)

• Improve chunking strategy

• Add metadata filtering

• Tune similarity threshold

Issue: Verification Too Strict/Lenient

Solution:

• Adjust system prompt guidelines

• Modify confidence thresholds

• Improve claim extraction

• Add claim type handling

Issue: High API Costs

Solution:

• Cache embeddings in database

• Use cheaper models for verification

• Reduce verification frequency

• Optimize chunk sizes

FINAL NOTES

Development Philosophy

• Build incrementally - each phase must work before moving to next

• Test thoroughly - verify success criteria at each milestone

• Commit often - tag each phase completion

• Document learning - note what works and what doesn't

• No rush - this is a learning journey, not a race

Learning Resources

As you progress through phases, you'll need to learn:

• Phase 2: Async Python, Pydantic-AI basics

• Phase 3: Web scraping, embeddings, vector databases

• Phase 4: LLM prompt engineering, fact verification

• Phase 5: Logging, monitoring, UX design

Recommended approach:

• Learn as you build (just-in-time learning)

• Use AI assistance (Claude, ChatGPT) for guidance

• Reference existing RAG implementations on GitHub

• Join communities (r/LangChain, r/LocalLLaMA)

Expected Timeline

Phase	**Duration**	**Cumulative**
Phase 2.0	2-4 weeks	2-4 weeks
Phase 3.0	4-6 weeks	6-10 weeks
Phase 4.0	4-6 weeks	10-16 weeks
Phase 5.0	2-4 weeks	12-20 weeks

Total: 3-5 months to complete truth verification system

Remember

You already built the hardest part - the foundation. Everything from here is filling in the pieces. Your architecture is solid. Your approach is right. Now it's just execution.

Take your time. Learn deeply. Build something you trust.