Claude Code CLI: Detailed Context Management Implementation Roadmap

Prerequisites & Initial Setup

```bash
# Navigate to your project root
cd /home/richardw/crypto_rf_trading_system/

# Create context management directory structure
mkdir -p .claude/{templates,scripts,contexts,metrics}
mkdir -p docs/claude/{modules,strategies,architecture}
```

Phase 1: Foundation Setup (Week 1-2)

Day 1-2: Create Master CLAUDE.md File

Create `/home/richardw/crypto_rf_trading_system/CLAUDE.md`:

```markdown
# ULTRATHINK Crypto Trading System

## System Overview
This is a production-grade algorithmic trading system with 100+ Python modules implementing institutional-level crypt

## Architecture Overview
- **Core Trading Engine**: Real-time order execution and management
- **Data Pipeline**: 99.5% validated market data from multiple sources
- **Risk Management**: Position limits, drawdown controls, exposure monitoring
- **Strategy Framework**: 15+ algorithmic strategies with ML optimization
- **Backtesting Engine**: Walk-forward validation with 51 windows

## Critical Performance Requirements
- Order Decision Latency: <10ms
- Data Processing: 10,000+ ticks/second
- Uptime Target: 99.99%
- Memory Usage: <4GB under normal operation

## Project Structure
```

```
crypto_rf_trading_system/
├── phase1/               # Foundation components (data, validation)
├── phase2/              # Advanced features (ML, optimization)
├── models/              # Trained RF models and configurations
├── strategies/          # Trading strategy implementations
├── execution/           # Order management and routing
├── risk/                # Risk management and controls
├── data/                # Market data handling
├── analytics/           # Performance analysis
├── ultrathink/          # Core reasoning engine
└── meta_optim/          # Hyperparameter optimization
```

```
## Key Dependencies
- Python 3.9+ (asyncio for real-time processing)
- NumPy/Pandas (vectorized calculations)
- Scikit-learn (Random Forest models)
- YFinance (market data)
- Custom ULTRATHINK framework

## Development Guidelines
1. All new features must maintain <10ms latency
2. Use type hints for all trading-critical functions
3. Implement comprehensive error handling for market operations
4. Document all strategy parameters and thresholds
5. Test with historical data before live deployment

## Current Trading Status
- Live Paper Trading: enhanced_paper_trader_24h.py
- Active Models: 4 RF models (entry/position/exit/profit)
- Session Logs: logs/enhanced_24hr_trading/

## Module Priority Levels
- **CRITICAL**: execution/, risk/, live trading scripts
- **HIGH**: strategies/, models/, data fetchers
- **MEDIUM**: analytics/, backtesting/, features/
- **LOW**: visualization/, reports/, demos/
```

Day 3-4: Create Module-Specific Context Files

Create context files for each major subsystem:

.claude/contexts/data_pipeline.md:

markdown

```
# Data Pipeline Context

## Overview
Handles real-time and historical market data ingestion, validation, and distribution.

## Critical Modules
- `data/data_fetcher.py` - Primary data interface
- `data/yfinance_fetcher.py` - Yahoo Finance integration
- `data/minute_data_manager.py` - High-frequency data handling
- `phase1/enhanced_data_collector.py` - 99.5% quality validation

## Key Functions
- `fetch_ohlcv_data()` - Get market data with validation
- `validate_data_quality()` - Ensure 99.5% data integrity
- `stream_realtime_data()` - WebSocket data streaming

## Performance Considerations
- Cache frequently accessed data in memory
- Use pandas vectorization for calculations
- Implement circuit breakers for data anomalies

## Common Issues & Solutions
- Missing data points: Use forward-fill with limits
- Network latency: Implement local buffering
- API rate limits: Use rotating credentials
```

`.claude/contexts/trading_strategies.md`:

markdown

```
# Trading Strategies Context

## Overview
Implementation of 15+ algorithmic trading strategies with ML optimization.

## Strategy Categories
1. **Momentum-based**: Trend following, breakout detection
2. **Mean Reversion**: Statistical arbitrage, pairs trading
3. **ML-Enhanced**: Random Forest signal generation
4. **Market Making**: Bid-ask spread capture

## Critical Strategy Modules
- `strategies/long_short_strategy.py` - Core long/short logic
- `strategies/minute_trading_strategies.py` - HFT strategies
- `enhanced_rf_ensemble.py` - ML ensemble predictions

## Strategy Parameters
- Momentum Threshold: 1.78% per hour
- Position Limits: Max 50% portfolio per position
- Stop Loss: -2% from entry
- Take Profit: Dynamic based on volatility

## Integration Points
- Signals feed into `execution/order_router.py`
- Risk checks via `risk/risk_manager.py`
- Performance tracking in `analytics/`
```

Day 5-6: Implement Smart .claudeignore

Create `/home/richardw/crypto_rf_trading_system/.claudeignore`:

```
gitignore
```

```
# Large data files
*.csv
*.xlsx
*.parquet
data/raw/*
data/processed/*
backtest_results/*

# Logs and temporary files
logs/
*.log
*.tmp
*.cache
__pycache__/
*.pyc
.pytest_cache/

# Virtual environments
venv/
.venv/
env/
.env/

# Build artifacts
build/
dist/
*.egg-info/

# Jupyter notebooks (unless specifically needed)
*.ipynb
.ipynb_checkpoints/

# Model binaries (reference by path when needed)
*.pkl
*.joblib
*.h5

# Documentation builds
docs/_build/
docs/generated/

# IDE files
.vscode/
.idea/
*.swp
*.swo
```

```
# Secrets and credentials
.env
secrets/
config/prod/
*_credentials.json
api_keys.py

# Test data
test_data/
fixtures/large/

# Historical data archives
historical_data/
market_data_archive/

# Performance reports (PDFs, images)
reports/*.pdf
reports/*.png
reports/*.jpg
```

Day 7-8: Create Context Loading Scripts

`.claude/scripts/load_context.py`:

```
python
```

```
#!/usr/bin/env python3
"""
Dynamic context loader for Claude Code CLI
Loads relevant context based on the current task
"""

import os
import sys
import json
import ast
from pathlib import Path
from typing import Dict, List, Set

class ContextLoader:
    def __init__(self, project_root: Path):
        self.project_root = project_root
        self.context_dir = project_root / ".claude" / "contexts"
        self.module_graph = self._build_module_graph()

    def _build_module_graph(self) -> Dict[str, Set[str]]:
        """Build dependency graph of Python modules"""
        graph = {}

        for py_file in self.project_root.rglob("*.py"):
            if any(part.startswith('.') for part in py_file.parts):
                continue

            module_name = str(py_file.relative_to(self.project_root))
            imports = self._extract_imports(py_file)
            graph[module_name] = imports

        return graph

    def _extract_imports(self, file_path: Path) -> Set[str]:
        """Extract imports from a Python file"""
        imports = set()

        try:
            with open(file_path, 'r') as f:
                tree = ast.parse(f.read())

            for node in ast.walk(tree):
                if isinstance(node, ast.Import):
                    for alias in node.names:
                        imports.add(alias.name)
                elif isinstance(node, ast.ImportFrom):
                    if node.module:
                        imports.add(node.module)

        except Exception:
            pass

        return imports

    def get_context_for_module(self, module_path: str) -> str:
        """Get relevant context for a specific module"""
        contexts = []

        # Add base context
        base_context = self.project_root / "CLAUDE.md"
        if base_context.exists():
            contexts.append(f"# Base Context\n{base_context.read_text()}\n")

        # Add module-specific context
        module_category = self._categorize_module(module_path)
        category_context = self.context_dir / f"{module_category}.md"

        if category_context.exists():
            contexts.append(f"# {module_category.title()} Context\n{category_context.read_text()}\n")

        # Add dependency contexts
        if module_path in self.module_graph:
            deps = self.module_graph[module_path]
            for dep in deps:
                if dep.startswith(('strategies', 'execution', 'risk', 'data')):
                    dep_context = self.context_dir / f"{dep.split('.')[0]}.md"
                    if dep_context.exists() and dep_context not in contexts:
                        contexts.append(f"# {dep.title()} Context\n{dep_context.read_text()}\n")

        return "\n---\n".join(contexts)

    def _categorize_module(self, module_path: str) -> str:
        """Categorize a module based on its path"""
        if module_path.startswith('strategies/'):
            return 'trading_strategies'
        elif module_path.startswith('execution/'):
            return 'order_execution'
        elif module_path.startswith('risk/'):
            return 'risk_management'
        elif module_path.startswith('data/'):
            return 'data_pipeline'
        elif module_path.startswith('models/'):
            return 'machine_learning'
        elif module_path.startswith('analytics/'):
            return 'performance_analytics'
        else:
            return 'general'

if __name__ == "__main__":
    if len(sys.argv) > 1:
        module = sys.argv[1]
        loader = ContextLoader(Path.cwd())
        context = loader.get_context_for_module(module)
        print(context)
```

Day 9-10: Create Context Templates

`.claude/templates/module_context_template.md`:

markdown

```markdown
# [Module Name] Context

## Overview
[Brief description of module purpose and role in the system]

## Critical Files
- `path/to/main.py` - [Description]
- `path/to/helper.py` - [Description]

## Key Classes and Functions
### ClassName
- Purpose: [What it does]
- Key Methods:
  - `method_name()` - [Description]
  - `another_method()` - [Description]

### function_name()
- Purpose: [What it does]
- Parameters: [Key parameters]
- Returns: [What it returns]
- Performance: [Any performance considerations]

## Dependencies
- Internal: [List of internal module dependencies]
- External: [List of external package dependencies]

## Configuration
- Environment Variables: [Any env vars used]
- Config Files: [Any config files referenced]
- Constants: [Important constants]

## Common Patterns
[Describe common usage patterns or workflows]

## Performance Considerations
- [Memory usage notes]
- [Latency requirements]
- [Optimization opportunities]

## Testing
- Test Files: `tests/test_module.py`
- Key Test Scenarios: [List important test cases]

## Known Issues & TODOs
```

- [ ] [Issue or improvement needed]
- [ ] [Another issue]

Phase 2: Advanced Context Optimization (Week 3-4)

Day 11-13: Implement Semantic Chunking System

`.claude/scripts/semantic_chunker.py`:

python

```python
#!/usr/bin/env python3
"""
Semantic chunking system for efficient context loading
Uses Tree-sitter for AST parsing and intelligent chunking
"""

import os
import json
import hashlib
from pathlib import Path
from typing import List, Dict, Tuple
from dataclasses import dataclass
import tree_sitter
from tree_sitter import Language, Parser

# You'll need to build the Python language library
# python3 -m pip install tree-sitter
# git clone https://github.com/tree-sitter/tree-sitter-python
# python3 build_parser.py  # Create this to build the .so file

@dataclass
class CodeChunk:
    """Represents a semantic chunk of code"""
    id: str
    file_path: str
    start_line: int
    end_line: int
    chunk_type: str  # 'class', 'function', 'module_docstring', etc.
    content: str
    dependencies: List[str]
    tokens: int

    def to_dict(self) -> Dict:
        return {
            'id': self.id,
            'file_path': self.file_path,
            'start_line': self.start_line,
            'end_line': self.end_line,
            'chunk_type': self.chunk_type,
            'dependencies': self.dependencies,
            'tokens': self.tokens
        }

class SemanticChunker:
    def __init__(self, project_root: Path):
        self.project_root = project_root
        self.chunks_dir = project_root / ".claude" / "chunks"
        self.chunks_dir.mkdir(exist_ok=True)

        # Initialize Tree-sitter
        PY_LANGUAGE = Language('build/python-languages.so', 'python')
        self.parser = Parser()
        self.parser.set_language(PY_LANGUAGE)

        self.chunks_index = {}

    def chunk_codebase(self) -> None:
        """Chunk entire codebase semantically"""
        for py_file in self.project_root.rglob("*.py"):
            if self._should_skip_file(py_file):
                continue

            chunks = self._chunk_file(py_file)
            self._save_chunks(py_file, chunks)

        self._save_index()

    def _should_skip_file(self, file_path: Path) -> bool:
        """Check if file should be skipped based on .claudeignore"""
        # Implementation would check against .claudeignore patterns
        skip_dirs = {'venv', '.venv', '__pycache__', 'test_data'}
        return any(part in skip_dirs for part in file_path.parts)

    def _chunk_file(self, file_path: Path) -> List[CodeChunk]:
        """Chunk a single file into semantic units"""
        chunks = []

        with open(file_path, 'rb') as f:
            content = f.read()

        tree = self.parser.parse(content)

        # Extract different types of chunks
        chunks.extend(self._extract_classes(tree, file_path, content))
        chunks.extend(self._extract_functions(tree, file_path, content))
        chunks.extend(self._extract_module_docstring(tree, file_path, content))

        return chunks

    def _extract_classes(self, tree, file_path: Path, content: bytes) -> List[CodeChunk]:
        """Extract class definitions as chunks"""
        chunks = []
```

```python
        class_query = self.parser.language.query("""
            (class_definition
                name: (identifier) @class_name
                body: (block) @class_body) @class
        """)

        captures = class_query.captures(tree.root_node)

        for node, _ in captures:
            if node.type == 'class_definition':
                chunk_content = content[node.start_byte:node.end_byte].decode('utf-8')
                chunk_id = hashlib.md5(chunk_content.encode()).hexdigest()[:8]

                chunk = CodeChunk(
                    id=chunk_id,
                    file_path=str(file_path.relative_to(self.project_root)),
                    start_line=node.start_point[0],
                    end_line=node.end_point[0],
                    chunk_type='class',
                    content=chunk_content,
                    dependencies=self._extract_dependencies(chunk_content),
                    tokens=len(chunk_content.split())  # Simple token count
                )
                chunks.append(chunk)

        return chunks

    def _extract_functions(self, tree, file_path: Path, content: bytes) -> List[CodeChunk]:
        """Extract function definitions as chunks"""
        chunks = []

        # Query for top-level functions (not inside classes)
        function_query = self.parser.language.query("""
            (module
                (function_definition
                    name: (identifier) @func_name) @function)
        """)

        captures = function_query.captures(tree.root_node)

        for node, _ in captures:
            if node.type == 'function_definition':
                # Check if this function is inside a class
                parent = node.parent
                is_top_level = True
                while parent:
                    if parent.type == 'class_definition':
                        is_top_level = False
                        break
                    parent = parent.parent

                if is_top_level:
                    chunk_content = content[node.start_byte:node.end_byte].decode('utf-8')
                    chunk_id = hashlib.md5(chunk_content.encode()).hexdigest()[:8]

                    chunk = CodeChunk(
                        id=chunk_id,
                        file_path=str(file_path.relative_to(self.project_root)),
                        start_line=node.start_point[0],
                        end_line=node.end_point[0],
                        chunk_type='function',
                        content=chunk_content,
                        dependencies=self._extract_dependencies(chunk_content),
                        tokens=len(chunk_content.split())
                    )
                    chunks.append(chunk)

        return chunks

    def _extract_module_docstring(self, tree, file_path: Path, content: bytes) -> List[CodeChunk]:
        """Extract module-level docstring and imports"""
        chunks = []

        # Get first statement if it's a docstring
        module = tree.root_node
        if module.type == 'module' and module.child_count > 0:
            first_child = module.child(0)
            if first_child.type == 'expression_statement':
                string_node = first_child.child(0)
                if string_node.type == 'string':
                    # This is a module docstring
                    # Include imports as well
                    import_end = 0
                    for child in module.children:
                        if child.type in ['import_statement', 'import_from_statement']:
                            import_end = child.end_byte
                        elif child.type not in ['expression_statement', 'comment']:
                            break

                    if import_end > 0:
                        chunk_content = content[0:import_end].decode('utf-8')
                    else:
                        chunk_content = content[string_node.start_byte:string_node.end_byte].decode('utf-8')
                chunk_id = hashlib.md5(chunk_content.encode()).hexdigest()[:8]

                chunk = CodeChunk(
                    id=chunk_id,
                    file_path=str(file_path.relative_to(self.project_root)),
                    start_line=0,
                    end_line=import_end // 80,  # Rough estimate
                    chunk_type='module_header',
                    content=chunk_content,
                    dependencies=[],
                    tokens=len(chunk_content.split())
                )
                chunks.append(chunk)

        return chunks

    def _extract_dependencies(self, code: str) -> List[str]:
        """Extract imported modules from code chunk"""
        dependencies = []
        lines = code.split('\n')

        for line in lines:
            line = line.strip()
            if line.startswith('import '):
                dep = line.split()[1].split('.')[0]
                dependencies.append(dep)
            elif line.startswith('from '):
                parts = line.split()
                if len(parts) >= 2:
                    dep = parts[1].split('.')[0]
                    dependencies.append(dep)

        return list(set(dependencies))

    def _save_chunks(self, file_path: Path, chunks: List[CodeChunk]) -> None:
        """Save chunks for a file"""
        rel_path = file_path.relative_to(self.project_root)
        chunk_file = self.chunks_dir / f"{rel_path.stem}_chunks.json"
        chunk_file.parent.mkdir(parents=True, exist_ok=True)

        chunk_data = {
            'file_path': str(rel_path),
            'chunks': [chunk.to_dict() for chunk in chunks]
        }

        with open(chunk_file, 'w') as f:
            json.dump(chunk_data, f, indent=2)
        # Update index
        self.chunks_index[str(rel_path)] = {
            'chunk_file': str(chunk_file.relative_to(self.project_root)),
            'chunk_count': len(chunks),
            'total_tokens': sum(c.tokens for c in chunks)
        }

    def _save_index(self) -> None:
        """Save the chunks index"""
        index_file = self.chunks_dir / "index.json"
        with open(index_file, 'w') as f:
            json.dump(self.chunks_index, f, indent=2)

if __name__ == "__main__":
    chunker = SemanticChunker(Path.cwd())
    print("Starting semantic chunking of codebase...")
    chunker.chunk_codebase()
    print(f"Chunking complete. Index saved to .claude/chunks/index.json")
```

Day 14-16: Implement CGRAG-Inspired Retrieval

.claude/scripts/cgrag_retrieval.py:

python

```python
#!/usr/bin/env python3
"""
CGRAG-inspired retrieval system for intelligent context loading
Two-stage process: concept identification, then context retrieval
"""

import json
import numpy as np
from pathlib import Path
from typing import List, Dict, Set, Tuple
from dataclasses import dataclass
import re
from collections import defaultdict

@dataclass
class Query:
    text: str
    intent: str  # 'debug', 'feature', 'optimization', 'analysis'
    concepts: List[str]
    modules_mentioned: List[str]

@dataclass
class RetrievalResult:
    chunks: List[Dict]
    total_tokens: int
    relevance_score: float

class CGRAGRetriever:
    def __init__(self, project_root: Path):
        self.project_root = project_root
        self.chunks_dir = project_root / ".claude" / "chunks"
        self.contexts_dir = project_root / ".claude" / "contexts"

        # Load chunks index
        with open(self.chunks_dir / "index.json", 'r') as f:
            self.chunks_index = json.load(f)

        # Define concept mappings for trading system
        self.concept_map = {
            'trading': ['strategies', 'execution', 'orders', 'positions'],
            'risk': ['risk_manager', 'position_limits', 'stop_loss', 'exposure'],
            'data': ['market_data', 'fetcher', 'validation', 'stream'],
            'backtest': ['backtesting', 'walk_forward', 'historical', 'validation'],
            'ml': ['random_forest', 'models', 'features', 'predictions'],
            'performance': ['analytics', 'metrics', 'pnl', 'sharpe'],
            'optimization': ['hyperparameter', 'genetic', 'meta_optim', 'tuning'],
            'realtime': ['async', 'websocket', 'streaming', 'latency']
        }

    def retrieve(self, query: str, max_tokens: int = 8000) -> RetrievalResult:
        """Two-stage retrieval process"""
        # Stage 1: Analyze query and identify concepts
        analyzed_query = self._analyze_query(query)

        # Stage 2: Retrieve relevant chunks based on concepts
        relevant_chunks = self._retrieve_chunks(analyzed_query, max_tokens)

        return relevant_chunks

    def _analyze_query(self, query: str) -> Query:
        """Analyze query to extract intent and concepts"""
        query_lower = query.lower()

        # Determine intent
        intent = 'general'
        if any(word in query_lower for word in ['debug', 'error', 'fix', 'issue']):
            intent = 'debug'
        elif any(word in query_lower for word in ['add', 'implement', 'create', 'new']):
            intent = 'feature'
        elif any(word in query_lower for word in ['optimize', 'improve', 'faster', 'performance']):
            intent = 'optimization'
        elif any(word in query_lower for word in ['analyze', 'report', 'metrics', 'performance']):
            intent = 'analysis'

        # Extract concepts
        concepts = []
        for concept, keywords in self.concept_map.items():
            if any(keyword in query_lower for keyword in keywords):
                concepts.append(concept)

        # Extract module references
        modules_mentioned = []
        # Look for file paths or module names
        module_pattern = r'(\w+/\w+\.py|\w+\.py|`\w+`)'
        matches = re.findall(module_pattern, query)
        modules_mentioned.extend([m.strip('`') for m in matches])

        return Query(
            text=query,
            intent=intent,
            concepts=concepts,
            modules_mentioned=modules_mentioned
        )
```

```python
def _retrieve_chunks(self, query: Query, max_tokens: int) -> RetrievalResult:
    """Retrieve relevant chunks based on analyzed query"""
    relevant_chunks = []
    seen_chunks = set()
    total_tokens = 0

    # Priority 1: Directly mentioned modules
    for module in query.modules_mentioned:
        chunks = self._get_chunks_for_module(module)
        for chunk in chunks:
            if chunk['id'] not in seen_chunks and total_tokens + chunk['tokens'] <= max_tokens:
                relevant_chunks.append(chunk)
                seen_chunks.add(chunk['id'])
                total_tokens += chunk['tokens']

    # Priority 2: Concept-based retrieval
    for concept in query.concepts:
        related_modules = self._get_modules_for_concept(concept)
        for module in related_modules:
            chunks = self._get_chunks_for_module(module)

            # For each module, prioritize based on query intent
            prioritized_chunks = self._prioritize_chunks(chunks, query.intent)

            for chunk in prioritized_chunks:
                if chunk['id'] not in seen_chunks and total_tokens + chunk['tokens'] <= max_tokens:
                    relevant_chunks.append(chunk)
                    seen_chunks.add(chunk['id'])
                    total_tokens += chunk['tokens']

    # Priority 3: Dependencies of retrieved chunks
    dependencies = self._get_dependencies(relevant_chunks)
    for dep_module in dependencies:
        if total_tokens >= max_tokens * 0.8:  # Leave some room
            break

        chunks = self._get_chunks_for_module(dep_module)
        # Only get module headers for dependencies
        header_chunks = [c for c in chunks if c.get('chunk_type') == 'module_header']

        for chunk in header_chunks:
            if chunk['id'] not in seen_chunks and total_tokens + chunk['tokens'] <= max_tokens:
                relevant_chunks.append(chunk)
                seen_chunks.add(chunk['id'])
                total_tokens += chunk['tokens']
        # Calculate relevance score
        relevance_score = self._calculate_relevance(relevant_chunks, query)

        return RetrievalResult(
            chunks=relevant_chunks,
            total_tokens=total_tokens,
            relevance_score=relevance_score
        )

    def _get_chunks_for_module(self, module_path: str) -> List[Dict]:
        """Get all chunks for a module"""
        chunks = []

        # Try different path formats
        possible_paths = [
            module_path,
            f"{module_path}.py" if not module_path.endswith('.py') else module_path,
            module_path.replace('/', '.')
        ]

        for path in possible_paths:
            if path in self.chunks_index:
                chunk_file = self.project_root / self.chunks_index[path]['chunk_file']
                if chunk_file.exists():
                    with open(chunk_file, 'r') as f:
                        data = json.load(f)
                        chunks.extend(data['chunks'])
                        break

        return chunks

    def _get_modules_for_concept(self, concept: str) -> List[str]:
        """Get modules related to a concept"""
        concept_module_map = {
            'trading': ['strategies/', 'execution/', 'enhanced_paper_trader_24h.py'],
            'risk': ['risk/', 'phase1/triple_barrier_labeling.py'],
            'data': ['data/', 'phase1/enhanced_data_collector.py'],
            'backtest': ['backtesting/', 'comprehensive_backtest.py'],
            'ml': ['models/', 'enhanced_rf_ensemble.py'],
            'performance': ['analytics/', 'trading_pattern_analyzer.py'],
            'optimization': ['meta_optim/', 'phase2/'],
            'realtime': ['minute_feature_engineering.py', 'enhanced_live_monitor.py']
        }

        modules = []
        if concept in concept_module_map:
            for pattern in concept_module_map[concept]:
                # Find all modules matching pattern
                for module_path in self.chunks_index.keys():
                    if pattern in module_path:
                        modules.append(module_path)

        return modules

    def _prioritize_chunks(self, chunks: List[Dict], intent: str) -> List[Dict]:
        """Prioritize chunks based on query intent"""
        if intent == 'debug':
            # Prioritize error handling, validation, logging
            priority_keywords = ['error', 'exception', 'validate', 'check', 'log']
        elif intent == 'feature':
            # Prioritize class definitions, main functions
            return sorted(chunks, key=lambda c: 0 if c.get('chunk_type') == 'class' else 1)
        elif intent == 'optimization':
            # Prioritize performance-critical sections
            priority_keywords = ['performance', 'optimize', 'cache', 'vectorize', 'parallel']
        elif intent == 'analysis':
            # Prioritize metrics, reporting functions
            priority_keywords = ['metric', 'analyze', 'report', 'calculate', 'measure']
        else:
            return chunks

        # Score chunks based on keyword presence
        scored_chunks = []
        for chunk in chunks:
            score = sum(1 for keyword in priority_keywords
                        if keyword in chunk.get('content', '').lower())
            scored_chunks.append((score, chunk))

        # Sort by score descending
        scored_chunks.sort(key=lambda x: x[0], reverse=True)
        return [chunk for _, chunk in scored_chunks]

    def _get_dependencies(self, chunks: List[Dict]) -> Set[str]:
        """Extract unique dependencies from chunks"""
        dependencies = set()

        for chunk in chunks:
            for dep in chunk.get('dependencies', []):
                # Map to internal modules if possible
                if dep in ['numpy', 'pandas', 'sklearn']:
                    continue  # Skip external deps

                # Try to find internal module
                for module_path in self.chunks_index.keys():
                    if dep in module_path:
                        dependencies.add(module_path)
                        break

        return dependencies

    def _calculate_relevance(self, chunks: List[Dict], query: Query) -> float:
        """Calculate relevance score for retrieved chunks"""
        if not chunks:
            return 0.0

        # Factors:
        # 1. Coverage of mentioned modules
        mentioned_coverage = 0
        if query.modules_mentioned:
            covered = sum(1 for m in query.modules_mentioned
                        if any(m in c.get('file_path', '') for c in chunks))
            mentioned_coverage = covered / len(query.modules_mentioned)

        # 2. Concept coverage
        concept_coverage = 0
        if query.concepts:
            covered_concepts = set()
            for chunk in chunks:
                for concept in query.concepts:
                    if concept in chunk.get('file_path', '').lower():
                        covered_concepts.add(concept)
            concept_coverage = len(covered_concepts) / len(query.concepts)

        # 3. Chunk type diversity
        chunk_types = set(c.get('chunk_type', 'unknown') for c in chunks)
        type_diversity = len(chunk_types) / 4  # Assume 4 main types

        # Weighted average
        weights = [0.5, 0.3, 0.2]
        scores = [mentioned_coverage, concept_coverage, type_diversity]

        relevance = sum(w * s for w, s in zip(weights, scores))
        return min(relevance, 1.0)

def format_context_for_claude(retrieval_result: RetrievalResult) -> str:
    """Format retrieval results for Claude"""
    output = []

    output.append(f"# Retrieved Context ({retrieval_result.total_tokens} tokens)")
    output.append(f"# Relevance Score: {retrieval_result.relevance_score:.2f}")
    output.append("")
```

````
    # Group chunks by file
    chunks_by_file = defaultdict(list)
    for chunk in retrieval_result.chunks:
        chunks_by_file[chunk['file_path']].append(chunk)

    # Format each file's chunks
    for file_path, file_chunks in chunks_by_file.items():
        output.append(f"## {file_path}")
        output.append("")

        # Sort chunks by line number
        file_chunks.sort(key=lambda c: c['start_line'])

        for chunk in file_chunks:
            output.append(f"### {chunk['chunk_type'].title()} (lines {chunk['start_line']}-{chunk['end_line']})")
            output.append("```python")
            output.append(chunk.get('content', ''))
            output.append("```")
            output.append("")

    return "\n".join(output)

if __name__ == "__main__":
    import sys

    if len(sys.argv) > 1:
        query = " ".join(sys.argv[1:])
        retriever = CGRAGRetriever(Path.cwd())
        result = retriever.retrieve(query)

        print(format_context_for_claude(result))
    else:
        print("Usage: python cgrag_retrieval.py 'your query here'")
````

Day 17-18: Set Up Monitoring and Metrics

`.claude/scripts/context_metrics.py`:

python

```python
#!/usr/bin/env python3
"""
Monitor and analyze context usage patterns for optimization
"""

import json
import sqlite3
from datetime import datetime
from pathlib import Path
from typing import Dict, List, Optional
import matplotlib.pyplot as plt
import pandas as pd

class ContextMetricsTracker:
    def __init__(self, project_root: Path):
        self.project_root = project_root
        self.metrics_dir = project_root / ".claude" / "metrics"
        self.metrics_dir.mkdir(exist_ok=True)

        # Initialize SQLite database for metrics
        self.db_path = self.metrics_dir / "context_metrics.db"
        self._init_database()

    def _init_database(self):
        """Initialize metrics database"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        # Create tables
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS context_usage (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
                query_text TEXT,
                query_intent TEXT,
                tokens_used INTEGER,
                chunks_retrieved INTEGER,
                relevance_score REAL,
                execution_time_ms INTEGER,
                modules_accessed TEXT,
                concepts_identified TEXT
            )
        """)

        cursor.execute("""
            CREATE TABLE IF NOT EXISTS module_access_frequency (
                module_path TEXT PRIMARY KEY,
                access_count INTEGER DEFAULT 0,
                total_tokens INTEGER DEFAULT 0,
                last_accessed DATETIME
            )
        """)

        cursor.execute("""
            CREATE TABLE IF NOT EXISTS performance_metrics (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
                metric_name TEXT,
                metric_value REAL,
                metadata TEXT
            )
        """)

        conn.commit()
        conn.close()

    def log_context_usage(self, query: str, retrieval_result: Dict,
                          execution_time_ms: int, query_analysis: Dict):
        """Log context usage for a query"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        # Log to context_usage table
        cursor.execute("""
            INSERT INTO context_usage
            (query_text, query_intent, tokens_used, chunks_retrieved,
             relevance_score, execution_time_ms, modules_accessed, concepts_identified)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?)
        """, (
            query,
            query_analysis.get('intent', 'unknown'),
            retrieval_result.get('total_tokens', 0),
            len(retrieval_result.get('chunks', [])),
            retrieval_result.get('relevance_score', 0.0),
            execution_time_ms,
            json.dumps(query_analysis.get('modules_mentioned', [])),
            json.dumps(query_analysis.get('concepts', []))
        ))

        # Update module access frequency
        modules = set()
        for chunk in retrieval_result.get('chunks', []):
            module = chunk.get('file_path', '')
            if module:
                modules.add(module)

        for module in modules:
            cursor.execute("""
                INSERT INTO module_access_frequency (module_path, access_count, last_accessed)
                VALUES (?, 1, CURRENT_TIMESTAMP)
                ON CONFLICT(module_path) DO UPDATE SET
                    access_count = access_count + 1,
                    last_accessed = CURRENT_TIMESTAMP
            """, (module,))

        conn.commit()
        conn.close()

    def generate_daily_report(self) -> Dict:
        """Generate daily metrics report"""
        conn = sqlite3.connect(self.db_path)

        # Load data into pandas
        df_usage = pd.read_sql_query("""
            SELECT * FROM context_usage
            WHERE date(timestamp) = date('now', 'localtime')
        """, conn)

        df_modules = pd.read_sql_query("""
            SELECT * FROM module_access_frequency
            ORDER BY access_count DESC
            LIMIT 20
        """, conn)

        conn.close()

        if df_usage.empty:
            return {"message": "No data for today"}

        report = {
            "date": datetime.now().strftime("%Y-%m-%d"),
            "total_queries": len(df_usage),
            "avg_tokens_per_query": df_usage['tokens_used'].mean(),
            "avg_chunks_per_query": df_usage['chunks_retrieved'].mean(),
            "avg_relevance_score": df_usage['relevance_score'].mean(),
            "avg_execution_time_ms": df_usage['execution_time_ms'].mean(),
            "query_intents": df_usage['query_intent'].value_counts().to_dict(),
            "top_accessed_modules": df_modules.head(10).to_dict('records')
        }
        # Save report
        report_path = self.metrics_dir / f"daily_report_{report['date']}.json"
        with open(report_path, 'w') as f:
            json.dump(report, f, indent=2)

        return report

    def visualize_metrics(self):
        """Create visualization of context usage patterns"""
        conn = sqlite3.connect(self.db_path)

        # Load recent data
        df_usage = pd.read_sql_query("""
            SELECT * FROM context_usage
            WHERE timestamp > datetime('now', '-7 days')
            ORDER BY timestamp
        """, conn)

        if df_usage.empty:
            print("No data to visualize")
            return

        # Convert timestamp to datetime
        df_usage['timestamp'] = pd.to_datetime(df_usage['timestamp'])

        # Create subplots
        fig, axes = plt.subplots(2, 2, figsize=(15, 10))
        fig.suptitle('Context Usage Metrics - Last 7 Days', fontsize=16)

        # 1. Token usage over time
        ax1 = axes[0, 0]
        df_usage.set_index('timestamp').resample('1H')['tokens_used'].mean().plot(ax=ax1)
        ax1.set_title('Average Tokens Used Per Hour')
        ax1.set_ylabel('Tokens')

        # 2. Query intent distribution
        ax2 = axes[0, 1]
        df_usage['query_intent'].value_counts().plot(kind='bar', ax=ax2)
        ax2.set_title('Query Intent Distribution')
        ax2.set_ylabel('Count')

        # 3. Relevance scores distribution
        ax3 = axes[1, 0]
        df_usage['relevance_score'].hist(bins=20, ax=ax3)
        ax3.set_title('Relevance Score Distribution')
        ax3.set_xlabel('Relevance Score')
        ax3.set_ylabel('Count')
```

```python
        # 4. Execution time vs tokens used
        ax4 = axes[1, 1]
        ax4.scatter(df_usage['tokens_used'], df_usage['execution_time_ms'], alpha=0.5)
        ax4.set_title('Execution Time vs Tokens Used')
        ax4.set_xlabel('Tokens Used')
        ax4.set_ylabel('Execution Time (ms)')

        plt.tight_layout()

        # Save plot
        plot_path = self.metrics_dir / f"metrics_visualization_{datetime.now().strftime('%Y%m%d')}.png"
        plt.savefig(plot_path, dpi=300, bbox_inches='tight')
        plt.close()

        print(f"Visualization saved to {plot_path}")

        # Also create module heatmap
        self._create_module_heatmap(conn)

        conn.close()

    def _create_module_heatmap(self, conn):
        """Create heatmap of module access patterns"""
        df_modules = pd.read_sql_query("""
            SELECT module_path, access_count,
                   strftime('%H', last_accessed) as hour,
                   strftime('%w', last_accessed) as day_of_week
            FROM module_access_frequency
            WHERE access_count > 5
        """, conn)

        if df_modules.empty:
            return

        # Pivot for heatmap
        pivot_data = df_modules.pivot_table(
            values='access_count',
            index='module_path',
            columns='hour',
            aggfunc='sum',
            fill_value=0
        )

        # Create heatmap
        plt.figure(figsize=(20, 10))
        plt.imshow(pivot_data.values, cmap='YlOrRd', aspect='auto')
        plt.colorbar(label='Access Count')

        # Labels
        plt.yticks(range(len(pivot_data.index)), pivot_data.index)
        plt.xticks(range(len(pivot_data.columns)), pivot_data.columns)
        plt.xlabel('Hour of Day')
        plt.ylabel('Module Path')
        plt.title('Module Access Patterns by Hour')

        plt.tight_layout()

        # Save
        heatmap_path = self.metrics_dir / f"module_heatmap_{datetime.now().strftime('%Y%m%d')}.png"
        plt.savefig(heatmap_path, dpi=300, bbox_inches='tight')
        plt.close()

    def get_optimization_recommendations(self) -> List[str]:
        """Generate optimization recommendations based on metrics"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        recommendations = []

        # 1. Check for frequently accessed modules that could be cached
        cursor.execute("""
            SELECT module_path, access_count, total_tokens
            FROM module_access_frequency
            WHERE access_count > 10
            ORDER BY access_count DESC
            LIMIT 5
        """)

        frequent_modules = cursor.fetchall()
        if frequent_modules:
            recommendations.append(
                f"Consider caching these frequently accessed modules: "
                f"{', '.join(m[0] for m in frequent_modules[:3])}"
            )

        # 2. Check for low relevance scores
        cursor.execute("""
            SELECT AVG(relevance_score) as avg_relevance
            FROM context_usage
            WHERE timestamp > datetime('now', '-24 hours')
        """)

        avg_relevance = cursor.fetchone()[0]
        if avg_relevance and avg_relevance < 0.5:
            recommendations.append(
                f"Low average relevance score ({avg_relevance:.2f}). "
                "Consider improving concept mapping or chunk indexing."
            )

        # 3. Check for high token usage
        cursor.execute("""
            SELECT AVG(tokens_used) as avg_tokens
            FROM context_usage
            WHERE timestamp > datetime('now', '-24 hours')
        """)

        avg_tokens = cursor.fetchone()[0]
        if avg_tokens and avg_tokens > 6000:
            recommendations.append(
                f"High average token usage ({avg_tokens:.0f}). "
                "Consider more aggressive filtering or smaller chunk sizes."
            )

        # 4. Check for slow queries
        cursor.execute("""
            SELECT COUNT(*) as slow_queries
            FROM context_usage
            WHERE execution_time_ms > 1000
            AND timestamp > datetime('now', '-24 hours')
        """)

        slow_queries = cursor.fetchone()[0]
        if slow_queries > 5:
            recommendations.append(
                f"Found {slow_queries} slow queries (>1s). "
                "Consider optimizing retrieval algorithm or using caching."
            )

        conn.close()

        return recommendations

if __name__ == "__main__":
    tracker = ContextMetricsTracker(Path.cwd())

    # Generate daily report
    report = tracker.generate_daily_report()
    print("Daily Report Generated:")
    print(json.dumps(report, indent=2))
```

```
# Create visualizations
tracker.visualize_metrics()

# Get recommendations
recommendations = tracker.get_optimization_recommendations()
print("\nOptimization Recommendations:")
for i, rec in enumerate(recommendations, 1):
    print(f"{i}. {rec}")
```

Phase 3: Production Optimization (Week 5-6)

Day 19-21: Trading-Specific Optimizations

`.claude/scripts/trading_context_optimizer.py`:

python

```
#!/usr/bin/env python3
"""
Trading-specific context optimizations for crypto trading system
"""

import json
from pathlib import Path
from typing import Dict, List, Set
from datetime import datetime
import asyncio

class TradingContextOptimizer:
    def __init__(self, project_root: Path):
        self.project_root = project_root

        # Define trading system priorities
        self.module_priorities = {
            'critical': {
                'execution/order_router.py',
                'risk/risk_manager.py',
                'enhanced_paper_trader_24h.py',
                'strategies/active_strategy.py'  # Currently active strategy
            },
            'high': {
                'data/market_data_stream.py',
                'models/random_forest_model.py',
                'analytics/real_time_metrics.py'
            },
            'medium': {
                'backtesting/',
                'features/',
                'visualization/'
            },
            'low': {
                'tests/',
                'demos/',
                'reports/'
            }
        }

        # Strategy-specific contexts
        self.strategy_contexts = {
            'momentum': [
                'strategies/momentum_strategy.py',
                'features/momentum_indicators.py',
                'analytics/trend_analysis.py'
            ],
            'mean_reversion': [
                'strategies/mean_reversion_strategy.py',
                'features/statistical_indicators.py',
                'analytics/spread_analysis.py'
            ],
            'ml_based': [
                'models/random_forest_model.py',
                'features/ultra_feature_engineering.py',
                'phase2/advanced_technical_indicators.py'
            ]
        }

    def create_trading_context(self, context_type: str) -> str:
        """Create specialized context for different trading scenarios"""

        if context_type == 'live_trading':
            return self._create_live_trading_context()
        elif context_type == 'strategy_development':
            return self._create_strategy_dev_context()
        elif context_type == 'risk_analysis':
            return self._create_risk_analysis_context()
        elif context_type == 'performance_analysis':
            return self._create_performance_context()
        else:
            return self._create_general_trading_context()

    def _create_live_trading_context(self) -> str:
        """Context optimized for live trading operations"""
        context = """# Live Trading Context

## Active Systems
- Paper Trading: enhanced_paper_trader_24h.py
- Monitoring: enhanced_live_monitor.py
- Risk Manager: risk/risk_manager.py

## Critical Paths
1. Market Data → Strategy Signal → Risk Check → Order Execution
2. Position Monitoring → Risk Limits → Auto-Liquidation

## Performance Requirements
- Order Decision: <10ms
- Risk Check: <5ms
- Data Processing: <2ms per tick

## Key Functions
- `execute_trade()` - Main trading logic
```

~~~
- `check_risk_limits()` - Position and exposure checks
- `stream_market_data()` - Real-time data handling

## Error Handling
- Network disconnection → Graceful reconnect with state recovery
- Invalid market data → Validation and rejection
- Risk limit breach → Immediate position reduction

## Current Configuration
```python
TRADING_CONFIG = {
    'max_position_size': 0.5,  # 50% of portfolio
    'stop_loss': 0.02,         # 2%
    'momentum_threshold': 0.0178,  # 1.78% per hour
    'min_volume': 1000000      # $1M daily volume
}
~~~

""""

return context

```
def _create_strategy_dev_context(self) -> str:
    """Context for strategy development"""
    context = """# Strategy Development Context
```

Strategy Framework

Base class: `strategies/base_strategy.py`

Required methods:

- `generate_signals()` - Return buy/sell signals
- `calculate_position_size()` - Risk-based sizing
- `get_strategy_params()` - Hyperparameters

Available Indicators

Technical: features/advanced_technical_indicators.py

- RSI, MACD, Bollinger Bands, Ichimoku
- Volume Profile, Order Flow Imbalance

On-chain: phase2/simulated_onchain_features.py

- MVRV, NVT, Exchange flows
- Whale activity metrics

Backtesting Pipeline

1. Data validation: phase1/enhanced_data_collector.py
2. Feature engineering: features/ultra_feature_engineering.py
3. Walk-forward testing: phase1/walk_forward_engine.py
4. Performance analysis: analytics/strategy_performance.py

Integration Points

- Signals → execution/signal_processor.py
- Risk checks → risk/position_calculator.py
- Performance tracking → analytics/trade_logger.py """ return context def _create_risk_analysis_context(self) -> str: """Context for risk analysis tasks""" context = """# Risk Analysis Context

Risk Management Framework

Core module: `risk/risk_manager.py`

Risk Metrics

- Position Risk: Max 50% portfolio per position
- Portfolio Risk: Max 20% daily VaR
- Correlation Risk: Monitor cross-asset correlations
- Liquidity Risk: Min $1M daily volume

Risk Controls

1. Pre-trade checks:
   - Position limits
   - Correlation limits
   - Liquidity requirements
2. Real-time monitoring:
   - Mark-to-market P&L
   - Exposure tracking
   - Drawdown monitoring
3. Auto-liquidation triggers:
   - Stop loss: -2% from entry
   - Portfolio drawdown: -5% daily
   - Correlation breach: >0.8

Risk Analytics

- VaR calculation: analytics/value_at_risk.py
- Stress testing: analytics/stress_test.py
- Risk attribution: analytics/risk_attribution.py """ return context def optimize_for_query_pattern(self, recent_queries: List[str]) -> Dict: """Analyze query patterns and optimize context loading""" patterns = { 'debugging': 0, 'feature_dev': 0, 'performance': 0, 'analysis': 0 }

```
for query in recent_queries:
    query_lower = query.lower()
    if any(word in query_lower for word in ['error', 'bug', 'fix', 'issue']):
        patterns['debugging'] += 1
    elif any(word in query_lower for word in ['add', 'implement', 'create']):
        patterns['feature_dev'] += 1
    elif any(word in query_lower for word in ['slow', 'optimize', 'performance']):
        patterns['performance'] += 1
    elif any(word in query_lower for word in ['analyze', 'report', 'metrics']):
        patterns['analysis'] += 1

# Recommend context optimizations
recommendations = []

dominant_pattern = max(patterns, key=patterns.get)

if dominant_pattern == 'debugging':
    recommendations.append("Preload error handling and logging modules")
    recommendations.append("Include detailed stack traces in context")
elif dominant_pattern == 'feature_dev':
    recommendations.append("Load base classes and interfaces")
    recommendations.append("Include integration test examples")
elif dominant_pattern == 'performance':
    recommendations.append("Load profiling results and benchmarks")
    recommendations.append("Include async/parallel processing examples")
elif dominant_pattern == 'analysis':
    recommendations.append("Load analytics modules and report templates")
    recommendations.append("Include data visualization examples")

return {
    'pattern_analysis': patterns,
    'dominant_pattern': dominant_pattern,
    'recommendations': recommendations
}
```

if **name** == "**main**": optimizer = TradingContextOptimizer(Path.cwd())

```
# Create different context types
contexts = {
    'live_trading': optimizer.create_trading_context('live_trading'),
    'strategy_dev': optimizer.create_trading_context('strategy_development'),
    'risk_analysis': optimizer.create_trading_context('risk_analysis')
}

# Save contexts
for name, context in contexts.items():
    context_path = Path.cwd() / ".claude" / "contexts" / f"{name}_context.md"
    context_path.parent.mkdir(parents=True, exist_ok=True)
    context_path.write_text(context)
    print(f"Created {name} context at {context_path}")
```

Day 22-23: Security Hardening

`.claude/scripts/secure_context_filter.py`:
```python
#!/usr/bin/env python3
"""
Security filter for removing sensitive information from context
"""

import re
import hashlib
from pathlib import Path
from typing import List, Dict, Set
import ast

class SecureContextFilter:
    def __init__(self, project_root: Path):
        self.project_root = project_root

        # Sensitive patterns to redact
        self.sensitive_patterns = [
            # API Keys and Secrets
            (r'api[_-]?key\s*=\s*["\']([^"\']+)["\']', 'API_KEY_REDACTED'),
            (r'secret[_-]?key\s*=\s*["\']([^"\']+)["\']', 'SECRET_REDACTED'),
            (r'password\s*=\s*["\']([^"\']+)["\']', 'PASSWORD_REDACTED'),

            # Exchange credentials
            (r'binance[_-]?api[_-]?key\s*=\s*["\']([^"\']+)["\']', 'EXCHANGE_KEY_REDACTED'),
            (r'exchange[_-]?secret\s*=\s*["\']([^"\']+)["\']', 'EXCHANGE_SECRET_REDACTED'),

            # Wallet addresses and private keys
            (r'0x[a-fA-F0-9]{40}', 'WALLET_ADDRESS_REDACTED'),
            (r'private[_-]?key\s*=\s*["\']([^"\']+)["\']', 'PRIVATE_KEY_REDACTED'),

            # URLs with embedded credentials
            (r'https?://[^:]+:([^@]+)@', 'https://USER:PASS_REDACTED@'),

            # Database connection strings
            (r'mongodb\+srv://[^"\']+', 'MONGODB_URL_REDACTED'),
            (r'postgresql://[^"\']+', 'POSTGRES_URL_REDACTED'),
        ]

        # Files that should never be included
        self.forbidden_files = {
            '.env',
        'secrets.py',
        'credentials.json',
        'api_keys.py',
        'private_keys.json'
    }

    # Secure module verification
    self.module_hashes = self._calculate_module_hashes()

def _calculate_module_hashes(self) -> Dict[str, str]:
    """Calculate hashes of critical modules for integrity checking"""
    hashes = {}
    critical_modules = [
        'execution/order_router.py',
        'risk/risk_manager.py',
        'strategies/base_strategy.py'
    ]

    for module in critical_modules:
        module_path = self.project_root / module
        if module_path.exists():
            with open(module_path, 'rb') as f:
                content = f.read()
                hashes[module] = hashlib.sha256(content).hexdigest()

    return hashes

def filter_content(self, content: str, file_path: str = None) -> str:
    """Filter sensitive information from content"""
    filtered_content = content

    # Check if file should be completely excluded
    if file_path and any(forbidden in file_path for forbidden in self.forbidden_files):
        return f"# FILE REDACTED: {file_path} contains sensitive information"

    # Apply pattern-based filtering
    for pattern, replacement in self.sensitive_patterns:
        filtered_content = re.sub(pattern, replacement, filtered_content, flags=re.IGNORECASE)

    # Additional filtering for specific file types
    if file_path and file_path.endswith('.py'):
        filtered_content = self._filter_python_code(filtered_content)

    return filtered_content

def _filter_python_code(self, code: str) -> str:
    """Additional filtering for Python code"""
```

```
        try:
            tree = ast.parse(code)

            class SensitiveNodeVisitor(ast.NodeVisitor):
                def __init__(self):
                    self.sensitive_assignments = []

                def visit_Assign(self, node):
                    # Check for sensitive variable assignments
                    for target in node.targets:
                        if isinstance(target, ast.Name):
                            var_name = target.id.lower()
                            if any(sensitive in var_name for sensitive in
                                   ['api_key', 'secret', 'password', 'private_key']):
                                self.sensitive_assignments.append(ast.get_source_segment(code, node))
                    self.generic_visit(node)

            visitor = SensitiveNodeVisitor()
            visitor.visit(tree)

            # Redact sensitive assignments
            for assignment in visitor.sensitive_assignments:
                if assignment:
                    code = code.replace(assignment, "# SENSITIVE_ASSIGNMENT_REDACTED")

        except:
            # If AST parsing fails, continue with pattern-based filtering
            pass

        return code

    def verify_module_integrity(self, module_path: str) -> bool:
        """Verify module hasn't been tampered with"""
        if module_path not in self.module_hashes:
            return True  # Not a critical module

        full_path = self.project_root / module_path
        if not full_path.exists():
            return False

        with open(full_path, 'rb') as f:
            content = f.read()
            current_hash = hashlib.sha256(content).hexdigest()

        return current_hash == self.module_hashes[module_path]

    def create_secure_context(self, original_context: str, context_metadata: Dict) -> str:
        """Create a secure version of the context"""
        secure_context = []

        # Add security header
        secure_context.append("# SECURE CONTEXT - Sensitive Information Redacted")
        secure_context.append(f"# Generated: {datetime.now().isoformat()}")
        secure_context.append("")

        # Filter the main context
        filtered_main = self.filter_content(original_context)
        secure_context.append(filtered_main)

        # Add integrity verification results
        if 'modules' in context_metadata:
            secure_context.append("\n## Module Integrity Verification")
            for module in context_metadata['modules']:
                if self.verify_module_integrity(module):
                    secure_context.append(f"✓ {module} - Verified")
                else:
                    secure_context.append(f"⚠ {module} - INTEGRITY CHECK FAILED")

        return "\n".join(secure_context)

    def scan_codebase_for_secrets(self) -> List[Dict]:
        """Scan codebase for potential secret leaks"""
        findings = []

        for py_file in self.project_root.rglob("*.py"):
            if any(part.startswith('.') for part in py_file.parts):
                continue

            try:
                content = py_file.read_text()

                # Check each pattern
                for pattern, _ in self.sensitive_patterns:
                    matches = re.finditer(pattern, content, re.IGNORECASE)
                    for match in matches:
                        findings.append({
                            'file': str(py_file.relative_to(self.project_root)),
                            'line': content[:match.start()].count('\n') + 1,
                            'pattern': pattern,
                            'severity': 'HIGH'
                        })

            except Exception as e:
                continue
        return findings

if __name__ == "__main__":
    filter = SecureContextFilter(Path.cwd())

    # Scan for secrets
    print("Scanning codebase for potential secrets...")
    findings = filter.scan_codebase_for_secrets()

    if findings:
        print(f"\n⚠️  Found {len(findings)} potential secret leaks:")
        for finding in findings[:10]:  # Show first 10
            print(f"  - {finding['file']}:{finding['line']} - {finding['pattern']}")
    else:
        print("✓ No potential secrets found")

    # Test filtering
    test_content = """
    API_KEY = "sk-1234567890abcdef"
    exchange_secret = "my-secret-key"

    def connect_to_exchange():
        client = BinanceClient(
            api_key="binance-key-12345",
            api_secret="binance-secret-67890"
        )
        return client
    """

    filtered = filter.filter_content(test_content, "test.py")
    print("\n--- Original ---")
    print(test_content)
    print("\n--- Filtered ---")
    print(filtered)
```

Day 24-26: Performance Tuning

.claude/scripts/context_cache_manager.py:

python

```python
#!/usr/bin/env python3
"""
Intelligent caching system for frequently accessed contexts
"""

import json
import time
import pickle
from pathlib import Path
from typing import Dict, List, Optional, Tuple
from datetime import datetime, timedelta
import hashlib
from dataclasses import dataclass
import asyncio
import aiofiles

@dataclass
class CacheEntry:
    key: str
    content: str
    tokens: int
    created_at: datetime
    last_accessed: datetime
    access_count: int
    ttl_seconds: int

    def is_expired(self) -> bool:
        return datetime.now() > self.created_at + timedelta(seconds=self.ttl_seconds)

class ContextCacheManager:
    def __init__(self, project_root: Path, max_cache_size_mb: int = 100):
        self.project_root = project_root
        self.cache_dir = project_root / ".claude" / "cache"
        self.cache_dir.mkdir(exist_ok=True)

        self.max_cache_size = max_cache_size_mb * 1024 * 1024  # Convert to bytes
        self.cache_index_path = self.cache_dir / "cache_index.json"
        self.cache_data_path = self.cache_dir / "cache_data.pkl"

        self.cache_index = self._load_cache_index()
        self.cache_data = self._load_cache_data()

        # Configuration
        self.ttl_config = {
            'static_modules': 86400,     # 24 hours for stable code
            'active_modules': 3600,      # 1 hour for actively developed code
            'query_results': 1800,       # 30 minutes for query results
            'chunk_combinations': 7200   # 2 hours for chunk combinations
        }

    def _load_cache_index(self) -> Dict:
        """Load cache index from disk"""
        if self.cache_index_path.exists():
            with open(self.cache_index_path, 'r') as f:
                return json.load(f)
        return {}

    def _load_cache_data(self) -> Dict[str, CacheEntry]:
        """Load cache data from disk"""
        if self.cache_data_path.exists():
            with open(self.cache_data_path, 'rb') as f:
                return pickle.load(f)
        return {}

    def _save_cache(self):
        """Save cache to disk"""
        # Save index
        with open(self.cache_index_path, 'w') as f:
            json.dump(self.cache_index, f, indent=2)

        # Save data
        with open(self.cache_data_path, 'wb') as f:
            pickle.dump(self.cache_data, f)

    def _calculate_cache_key(self, content_type: str, identifier: str) -> str:
        """Generate cache key"""
        combined = f"{content_type}:{identifier}"
        return hashlib.md5(combined.encode()).hexdigest()

    def _get_cache_size(self) -> int:
        """Calculate current cache size in bytes"""
        total_size = 0
        for entry in self.cache_data.values():
            # Rough estimate: 1 token ≈ 4 bytes
            total_size += entry.tokens * 4
        return total_size

    def _evict_lru(self, required_space: int):
        """Evict least recently used entries to make space"""
        current_size = self._get_cache_size()

        if current_size + required_space <= self.max_cache_size:
            return
        # Sort by last accessed time
        sorted_entries = sorted(
            self.cache_data.items(),
            key=lambda x: x[1].last_accessed
        )

        freed_space = 0
        for key, entry in sorted_entries:
            if current_size + required_space - freed_space <= self.max_cache_size:
                break

            freed_space += entry.tokens * 4
            del self.cache_data[key]
            if key in self.cache_index:
                del self.cache_index[key]

    async def get_cached_context(self, content_type: str, identifier: str) -> Optional[str]:
        """Retrieve context from cache if available"""
        cache_key = self._calculate_cache_key(content_type, identifier)

        if cache_key in self.cache_data:
            entry = self.cache_data[cache_key]

            # Check if expired
            if entry.is_expired():
                del self.cache_data[cache_key]
                del self.cache_index[cache_key]
                self._save_cache()
                return None

            # Update access info
            entry.last_accessed = datetime.now()
            entry.access_count += 1

            # Promote frequently accessed items
            if entry.access_count > 10:
                entry.ttl_seconds = int(entry.ttl_seconds * 1.5)

            self._save_cache()
            return entry.content

        return None

    async def cache_context(self, content_type: str, identifier: str,
                          content: str, tokens: int):
        """Cache context with appropriate TTL"""
```

```python
        cache_key = self._calculate_cache_key(content_type, identifier)

        # Determine TTL based on content type
        if content_type == 'module':
            # Check if it's an actively developed module
            module_path = self.project_root / identifier
            if module_path.exists():
                mtime = datetime.fromtimestamp(module_path.stat().st_mtime)
                if datetime.now() - mtime < timedelta(hours=24):
                    ttl = self.ttl_config['active_modules']
                else:
                    ttl = self.ttl_config['static_modules']
            else:
                ttl = self.ttl_config['static_modules']
        else:
            ttl = self.ttl_config.get(content_type, 3600)

        # Check cache size and evict if necessary
        required_space = tokens * 4
        self._evict_lru(required_space)

        # Create cache entry
        entry = CacheEntry(
            key=cache_key,
            content=content,
            tokens=tokens,
            created_at=datetime.now(),
            last_accessed=datetime.now(),
            access_count=1,
            ttl_seconds=ttl
        )

        self.cache_data[cache_key] = entry
        self.cache_index[cache_key] = {
            'content_type': content_type,
            'identifier': identifier,
            'tokens': tokens,
            'created_at': entry.created_at.isoformat()
        }

        self._save_cache()

    def get_cache_stats(self) -> Dict:
        """Get cache statistics"""
        total_entries = len(self.cache_data)
        total_size = self._get_cache_size()
        # Calculate hit rate
        total_accesses = sum(e.access_count for e in self.cache_data.values())

        # Group by content type
        type_stats = {}
        for key, index_entry in self.cache_index.items():
            content_type = index_entry['content_type']
            if content_type not in type_stats:
                type_stats[content_type] = {
                    'count': 0,
                    'total_tokens': 0,
                    'avg_access_count': 0
                }

            type_stats[content_type]['count'] += 1
            type_stats[content_type]['total_tokens'] += index_entry['tokens']

            if key in self.cache_data:
                type_stats[content_type]['avg_access_count'] += self.cache_data[key].access_count

        # Calculate averages
        for content_type, stats in type_stats.items():
            if stats['count'] > 0:
                stats['avg_access_count'] /= stats['count']

        return {
            'total_entries': total_entries,
            'total_size_mb': total_size / (1024 * 1024),
            'cache_utilization': (total_size / self.max_cache_size) * 100,
            'total_accesses': total_accesses,
            'type_statistics': type_stats,
            'most_accessed': self._get_most_accessed_entries(5)
        }

    def _get_most_accessed_entries(self, limit: int) -> List[Dict]:
        """Get most frequently accessed cache entries"""
        sorted_entries = sorted(
            self.cache_data.items(),
            key=lambda x: x[1].access_count,
            reverse=True
        )[:limit]

        results = []
        for key, entry in sorted_entries:
            if key in self.cache_index:
                results.append({
                    'identifier': self.cache_index[key]['identifier'],
                    'content_type': self.cache_index[key]['content_type'],
                    'access_count': entry.access_count,
                    'tokens': entry.tokens
                })

        return results

    def clear_expired(self):
        """Clear all expired cache entries"""
        expired_keys = []

        for key, entry in self.cache_data.items():
            if entry.is_expired():
                expired_keys.append(key)

        for key in expired_keys:
            del self.cache_data[key]
            if key in self.cache_index:
                del self.cache_index[key]

        if expired_keys:
            self._save_cache()

        return len(expired_keys)

    async def warm_cache(self, frequently_used_modules: List[str]):
        """Pre-warm cache with frequently used modules"""
        print("Warming cache with frequently used modules...")

        for module_path in frequently_used_modules:
            full_path = self.project_root / module_path
            if full_path.exists():
                # Check if already cached
                cached = await self.get_cached_context('module', module_path)
                if cached:
                    continue

                # Load and cache
                content = full_path.read_text()
                tokens = len(content.split())  # Simple estimation

                await self.cache_context('module', module_path, content, tokens)
                print(f"  ✓ Cached {module_path} ({tokens} tokens)")

class IncrementalContextUpdater:
    """Handles incremental updates to cached contexts"""
    def __init__(self, cache_manager: ContextCacheManager):
        self.cache_manager = cache_manager
        self.project_root = cache_manager.project_root

    async def update_changed_modules(self) -> List[str]:
        """Update cache for modules that have changed"""
        updated = []

        for key, index_entry in self.cache_manager.cache_index.items():
            if index_entry['content_type'] != 'module':
                continue

            module_path = index_entry['identifier']
            full_path = self.project_root / module_path

            if full_path.exists():
                # Check if file has been modified
                mtime = datetime.fromtimestamp(full_path.stat().st_mtime)
                cached_time = datetime.fromisoformat(index_entry['created_at'])

                if mtime > cached_time:
                    # Re-cache the module
                    content = full_path.read_text()
                    tokens = len(content.split())

                    await self.cache_manager.cache_context(
                        'module', module_path, content, tokens
                    )
                    updated.append(module_path)

        return updated

async def main():
    """Example usage and performance testing"""
    cache_manager = ContextCacheManager(Path.cwd())
    updater = IncrementalContextUpdater(cache_manager)

    # Warm cache with critical modules
    critical_modules = [
        'execution/order_router.py',
        'risk/risk_manager.py',
        'strategies/long_short_strategy.py',
        'data/data_fetcher.py'
    ]

    await cache_manager.warm_cache(critical_modules)
```

```python
    # Simulate cache usage
    print("\nSimulating cache usage...")

    # Test cache hit
    start_time = time.time()
    cached_content = await cache_manager.get_cached_context('module', 'execution/order_router.py')
    cache_time = time.time() - start_time

    if cached_content:
        print(f"Cache hit! Retrieved in {cache_time*1000:.2f}ms")

    # Clear expired entries
    expired_count = cache_manager.clear_expired()
    print(f"\nCleared {expired_count} expired entries")

    # Update changed modules
    updated = await updater.update_changed_modules()
    if updated:
        print(f"\nUpdated {len(updated)} changed modules:")
        for module in updated:
            print(f"  - {module}")

    # Show cache statistics
    stats = cache_manager.get_cache_stats()
    print("\nCache Statistics:")
    print(f"  Total entries: {stats['total_entries']}")
    print(f"  Total size: {stats['total_size_mb']:.2f} MB")
    print(f"  Cache utilization: {stats['cache_utilization']:.1f}%")
    print(f"  Total accesses: {stats['total_accesses']}")

    print("\nMost accessed entries:")
    for entry in stats['most_accessed']:
        print(f"  - {entry['identifier']} ({entry['access_count']} accesses)")

if __name__ == "__main__":
    asyncio.run(main())
```

Integration & Automation Scripts

Master Context Management Script

Create `.claude/claude_context_manager.py`:

python

```python
#!/usr/bin/env python3
"""
Master script for Claude Code context management
Integrates all context optimization components
"""

import sys
import argparse
import asyncio
from pathlib import Path
from datetime import datetime

# Import all components
sys.path.append(str(Path(__file__).parent))

from scripts.semantic_chunker import SemanticChunker
from scripts.cgrag_retrieval import CGRAGRetriever, format_context_for_claude
from scripts.context_metrics import ContextMetricsTracker
from scripts.trading_context_optimizer import TradingContextOptimizer
from scripts.secure_context_filter import SecureContextFilter
from scripts.context_cache_manager import ContextCacheManager, IncrementalContextUpdater

class ClaudeContextManager:
    def __init__(self, project_root: Path):
        self.project_root = project_root

        # Initialize all components
        self.chunker = SemanticChunker(project_root)
        self.retriever = CGRAGRetriever(project_root)
        self.metrics = ContextMetricsTracker(project_root)
        self.trading_optimizer = TradingContextOptimizer(project_root)
        self.security_filter = SecureContextFilter(project_root)
        self.cache_manager = ContextCacheManager(project_root)
        self.cache_updater = IncrementalContextUpdater(self.cache_manager)

    async def setup(self):
        """Initial setup for context management"""
        print("🚀 Setting up Claude Code context management...")

        # 1. Create directory structure
        dirs = [
            ".claude/templates",
            ".claude/scripts",
            ".claude/contexts",
            ".claude/metrics",
            ".claude/chunks",
            ".claude/cache"
        ]

        for dir_path in dirs:
            (self.project_root / dir_path).mkdir(parents=True, exist_ok=True)

        # 2. Check for CLAUDE.md
        claude_md_path = self.project_root / "CLAUDE.md"
        if not claude_md_path.exists():
            print("❌ CLAUDE.md not found! Please create it first.")
            return False

        # 3. Chunk the codebase
        print("\n📊 Chunking codebase...")
        self.chunker.chunk_codebase()

        # 4. Create trading-specific contexts
        print("\n📝 Creating trading-specific contexts...")
        for context_type in ['live_trading', 'strategy_development', 'risk_analysis']:
            context = self.trading_optimizer.create_trading_context(context_type)
            path = self.project_root / ".claude" / "contexts" / f"{context_type}.md"
            path.write_text(context)

        # 5. Warm up cache
        print("\n🔥 Warming up cache...")
        critical_modules = [
            'enhanced_paper_trader_24h.py',
            'execution/order_router.py',
            'risk/risk_manager.py',
            'strategies/long_short_strategy.py'
        ]
        await self.cache_manager.warm_cache(critical_modules)

        print("\n✅ Setup complete!")
        return True

    async def query(self, query_text: str, max_tokens: int = 8000):
        """Process a query and return optimized context"""
        start_time = datetime.now()

        # 1. Retrieve context
        result = self.retriever.retrieve(query_text, max_tokens)

        # 2. Format for Claude
        formatted_context = format_context_for_claude(result)

        # 3. Apply security filtering
```

```python
        secure_context = self.security_filter.filter_content(formatted_context)

        # 4. Log metrics
        execution_time = int((datetime.now() - start_time).total_seconds() * 1000)
        query_analysis = self.retriever._analyze_query(query_text)

        self.metrics.log_context_usage(
            query_text,
            {
                'chunks': result.chunks,
                'total_tokens': result.total_tokens,
                'relevance_score': result.relevance_score
            },
            execution_time,
            {
                'intent': query_analysis.intent,
                'concepts': query_analysis.concepts,
                'modules_mentioned': query_analysis.modules_mentioned
            }
        )

        return secure_context

    def generate_report(self):
        """Generate context usage report"""
        report = self.metrics.generate_daily_report()
        self.metrics.visualize_metrics()

        recommendations = self.metrics.get_optimization_recommendations()

        print("\n📊 Context Usage Report")
        print("=" * 50)
        print(f"Date: {report.get('date', 'N/A')}")
        print(f"Total queries: {report.get('total_queries', 0)}")
        print(f"Avg tokens/query: {report.get('avg_tokens_per_query', 0):.0f}")
        print(f"Avg relevance: {report.get('avg_relevance_score', 0):.2f}")

        print("\n🎯 Optimization Recommendations:")
        for i, rec in enumerate(recommendations, 1):
            print(f"{i}. {rec}")

    async def update_cache(self):
        """Update cache for changed modules"""
        print("🔄 Updating cache for changed modules...")
        updated = await self.cache_updater.update_changed_modules()

        if updated:
            print(f"Updated {len(updated)} modules:")
            for module in updated[:5]:  # Show first 5
                print(f"  - {module}")
        else:
            print("No modules needed updating")

        # Clear expired entries
        expired = self.cache_manager.clear_expired()
        if expired:
            print(f"Cleared {expired} expired cache entries")

async def main():
    parser = argparse.ArgumentParser(
        description="Claude Code Context Management System"
    )

    parser.add_argument(
        'command',
        choices=['setup', 'query', 'report', 'update-cache'],
        help='Command to execute'
    )

    parser.add_argument(
        '--query',
        type=str,
        help='Query text (for query command)'
    )

    parser.add_argument(
        '--max-tokens',
        type=int,
        default=8000,
        help='Maximum tokens for context'
    )

    args = parser.parse_args()

    # Initialize manager
    manager = ClaudeContextManager(Path.cwd())

    # Execute command
    if args.command == 'setup':
        await manager.setup()

    elif args.command == 'query':
        if not args.query:
            print("Error: --query required for query command")
            sys.exit(1)

        context = await manager.query(args.query, args.max_tokens)
        print(context)

    elif args.command == 'report':
        manager.generate_report()

    elif args.command == 'update-cache':
        await manager.update_cache()

if __name__ == "__main__":
    asyncio.run(main())
```

Quick Start Commands

bash

```bash
# Initial setup
cd /home/richardw/crypto_rf_trading_system/
python3 .claude/claude_context_manager.py setup

# Query with optimized context
python3 .claude/claude_context_manager.py query --query "How do I optimize the order execution latency?"

# Generate daily report
python3 .claude/claude_context_manager.py report

# Update cache for changed files
python3 .claude/claude_context_manager.py update-cache

# Direct usage in Claude Code CLI
claude "Implement a new momentum strategy" --context-script .claude/claude_context_manager.py
```

Monitoring Dashboard

Create `.claude/scripts/context_dashboard.py` for real-time monitoring:

python

```
#!/usr/bin/env python3
"""
Simple web dashboard for context management monitoring
"""

from flask import Flask, render_template_string, jsonify
import json
from pathlib import Path
from datetime import datetime

app = Flask(__name__)

DASHBOARD_HTML = """
<!DOCTYPE html>
<html>
<head>
    <title>Claude Context Dashboard</title>
    <style>
        body { font-family: Arial, sans-serif; margin: 20px; }
        .metric {
            display: inline-block;
            margin: 10px;
            padding: 20px;
            background: #f0f0f0;
            border-radius: 5px;
        }
        .metric h3 { margin: 0 0 10px 0; }
        .metric .value { font-size: 2em; font-weight: bold; }
        table { border-collapse: collapse; width: 100%; margin-top: 20px; }
        th, td { border: 1px solid #ddd; padding: 8px; text-align: left; }
        th { background-color: #4CAF50; color: white; }
    </style>
</head>
<body>
    <h1>Claude Context Management Dashboard</h1>

    <div id="metrics">
        <div class="metric">
            <h3>Cache Hit Rate</h3>
            <div class="value">{{ cache_stats.hit_rate }}%</div>
        </div>
        <div class="metric">
            <h3>Avg Token Usage</h3>
            <div class="value">{{ avg_tokens }}</div>
        </div>
        <div class="metric">
```

```
            <h3>Cache Size</h3>
            <div class="value">{{ cache_stats.size_mb }} MB</div>
        </div>
    </div>

    <h2>Most Accessed Modules</h2>
    <table>
        <tr>
            <th>Module</th>
            <th>Access Count</th>
            <th>Last Accessed</th>
        </tr>
        {% for module in top_modules %}
        <tr>
            <td>{{ module.path }}</td>
            <td>{{ module.count }}</td>
            <td>{{ module.last_accessed }}</td>
        </tr>
        {% endfor %}
    </table>

    <script>
        // Auto-refresh every 30 seconds
        setTimeout(() => location.reload(), 30000);
    </script>
</body>
</html>
"""

@app.route('/')
def dashboard():
    # Load metrics
    project_root = Path.cwd()
    metrics_db = project_root / ".claude" / "metrics" / "context_metrics.db"

    # Dummy data for example - replace with actual DB queries
    cache_stats = {
        'hit_rate': 85.3,
        'size_mb': 42.7
    }

    avg_tokens = 3456

    top_modules = [
        {'path': 'execution/order_router.py', 'count': 156, 'last_accessed': '5 min ago'},
        {'path': 'risk/risk_manager.py', 'count': 143, 'last_accessed': '12 min ago'},
        {'path': 'strategies/momentum_strategy.py', 'count': 98, 'last_accessed': '1 hour ago'}
    ]

    return render_template_string(
        DASHBOARD_HTML,
        cache_stats=cache_stats,
        avg_tokens=avg_tokens,
        top_modules=top_modules
    )

if __name__ == '__main__':
    app.run(debug=True, port=5555)
```

This comprehensive roadmap provides everything you need to implement sophisticated context management for your crypto trading system with Claude Code CLI. The system will dramatically reduce token usage while improving the relevance and accuracy of Claude's responses to your queries.