# AZ400 - Code Archaeologist & Feature Discovery Agent

## Comprehensive Code Parsing, Indexing & Lost Feature Recovery

---

## 🎯 Agent Profile

yaml

```yaml
Agent_ID: AZ400
Agent_Name: "Code Archaeologist"
Classification: S-Tier_Intelligence_Synthesis
Agent_Class: Meta-Discovery
Vault_Role: "The Digital Archaeologist who excavates buried functionality and creates comprehensive maps of all code a

Core_Mission: |
  Comprehensive discovery, parsing, indexing, and cross-referencing of ALL code
  components across the entire project ecosystem - including embedded code in
  documentation, PDFs, markdown files, comments, and any other format where
  functionality might be hidden or poorly referenced.

Specialization_Domains:
  - Multi-format code extraction and parsing
  - Comprehensive component indexing and cataloging
  - Cross-reference mapping and dependency analysis
  - Lost functionality discovery and recovery
  - Feature inventory management and optimization
  - Dead code detection vs. unreferenced valuable code
  - Documentation-embedded code archaeology
  - Legacy system component discovery
```

---

## 🔍 Comprehensive Parsing Engine

### Multi-Format Code Extraction

#### Universal File Parser

python

```python
class UniversalCodeParser:
    """Extracts code from any file format with embedded functionality"""

    def __init__(self):
        self.supported_formats = {
            # Direct code files
            "python": [".py", ".pyx", ".pyi"],
            "javascript": [".js", ".jsx", ".ts", ".tsx", ".mjs"],
            "html": [".html", ".htm", ".vue", ".svelte"],
            "css": [".css", ".scss", ".sass", ".less"],
            "sql": [".sql", ".psql", ".mysql"],
            "shell": [".sh", ".bash", ".zsh", ".fish"],
            "docker": ["Dockerfile", ".dockerfile"],
            "yaml": [".yml", ".yaml"],
            "json": [".json", ".jsonc"],

            # Documentation with embedded code
            "markdown": [".md", ".mdx", ".markdown"],
            "restructured_text": [".rst"],
            "jupyter": [".ipynb"],

            # Office documents
            "pdf": [".pdf"],
            "word": [".docx", ".doc"],
            "excel": [".xlsx", ".xls"],
            "powerpoint": [".pptx", ".ppt"],

            # Specialized formats
            "confluence": [".confluence"],
            "notion": [".notion"],
            "obsidian": [".obsidian"],
            "readme": ["README", "readme"],

            # Configuration files
            "config": [".env", ".config", ".ini", ".conf", ".toml"],
            "package": ["package.json", "requirements.txt", "Pipfile", "pyproject.toml"],

            # Version control
            "git": [".gitignore", ".gitmodules", ".git/hooks/*"],

            # CI/CD
            "cicd": [".github/workflows/*", ".gitlab-ci.yml", "azure-pipelines.yml"]
        }

        self.code_extractors = {
            format_type: self._create_extractor(format_type)
```

```
        for format_type in self.supported_formats.keys()
    }

async def parse_project_comprehensively(self, project_root):
    """Comprehensive parsing of entire project including all embedded code"""

    # Discover all files
    all_files = await self.discover_all_files(project_root)

    # Categorize by format
    categorized_files = await self.categorize_files_by_format(all_files)

    # Extract code from each category
    extraction_results = {}
    for format_type, files in categorized_files.items():
        extraction_results[format_type] = await self.extract_code_from_format(
            format_type, files
        )

    # Consolidate and cross-reference
    comprehensive_inventory = await self.consolidate_extraction_results(extraction_results)

    return comprehensive_inventory

async def extract_code_from_markdown(self, file_path):
    """Extract all code blocks, inline code, and embedded functionality from markdown"""

    with open(file_path, 'r', encoding='utf-8') as f:
        content = f.read()

    code_extractions = []

    # Extract fenced code blocks
    fenced_blocks = re.findall(r'```(\w+)?\n(.*?)\n```', content, re.DOTALL)
    for language, code in fenced_blocks:
        code_extractions.append({
            "type": "fenced_code_block",
            "language": language or "unknown",
            "code": code.strip(),
            "file_path": file_path,
            "extraction_method": "regex_fenced_block"
        })

    # Extract indented code blocks
    indented_blocks = re.findall(r'(?:^    .*$\n?)+', content, re.MULTILINE)
    for code_block in indented_blocks:
        code_extractions.append({
```

```
                "type": "indented_code_block",
                "language": "unknown",
                "code": code_block.strip(),
                "file_path": file_path,
                "extraction_method": "regex_indented_block"
            })

        # Extract inline code
        inline_code = re.findall(r'`([^`]+)`', content)
        for code in inline_code:
            if len(code) > 10:  # Only capture substantial inline code
                code_extractions.append({
                    "type": "inline_code",
                    "language": "unknown",
                    "code": code,
                    "file_path": file_path,
                    "extraction_method": "regex_inline_code"
                })

        # Extract command line examples
        command_blocks = re.findall(r'\$ (.+?)(?:\n|$)', content)
        for command in command_blocks:
            code_extractions.append({
                "type": "command_line",
                "language": "shell",
                "code": command,
                "file_path": file_path,
                "extraction_method": "regex_command_line"
            })

        # Extract function/method signatures mentioned in text
        function_signatures = re.findall(r'(\w+\([^)]*\))', content)
        for signature in function_signatures:
            code_extractions.append({
                "type": "function_signature",
                "language": "unknown",
                "code": signature,
                "file_path": file_path,
                "extraction_method": "regex_function_signature"
            })

        return code_extractions

    async def extract_code_from_pdf(self, file_path):
        """Extract code from PDF documents using multiple methods"""

        code_extractions = []
```

```
    try:
        import PyPDF2
        import pdfplumber

        # Method 1: PyPDF2 text extraction
        pdf_text = await self.extract_pdf_text_pypdf2(file_path)
        if pdf_text:
            text_code = await self.extract_code_from_text(pdf_text, file_path, "pdf_text")
            code_extractions.extend(text_code)

        # Method 2: pdfplumber for better formatting
        plumber_text = await self.extract_pdf_text_pdfplumber(file_path)
        if plumber_text:
            plumber_code = await self.extract_code_from_text(plumber_text, file_path, "pdf_plumber")
            code_extractions.extend(plumber_code)

        # Method 3: OCR if necessary (for scanned PDFs)
        if len(code_extractions) == 0:
            ocr_text = await self.extract_pdf_text_ocr(file_path)
            if ocr_text:
                ocr_code = await self.extract_code_from_text(ocr_text, file_path, "pdf_ocr")
                code_extractions.extend(ocr_code)

    except ImportError:
        print(f"PDF parsing libraries not available for {file_path}")

    return code_extractions

async def extract_code_from_jupyter(self, file_path):
    """Extract all code cells from Jupyter notebooks"""

    import json

    with open(file_path, 'r', encoding='utf-8') as f:
        notebook = json.load(f)

    code_extractions = []

    for cell_idx, cell in enumerate(notebook.get('cells', [])):
        if cell.get('cell_type') == 'code':
            source = cell.get('source', [])
            if isinstance(source, list):
                code = ''.join(source)
            else:
                code = source
```

```
                code_extractions.append({
                    "type": "jupyter_code_cell",
                    "language": "python",  # Default assumption
                    "code": code,
                    "file_path": file_path,
                    "cell_index": cell_idx,
                    "extraction_method": "jupyter_cell_parse"
                })

        return code_extractions

    async def extract_code_from_office_documents(self, file_path):
        """Extract code from Word, Excel, PowerPoint documents"""

        code_extractions = []
        file_extension = os.path.splitext(file_path)[1].lower()

        try:
            if file_extension in ['.docx', '.doc']:
                code_extractions = await self.extract_code_from_word(file_path)
            elif file_extension in ['.xlsx', '.xls']:
                code_extractions = await self.extract_code_from_excel(file_path)
            elif file_extension in ['.pptx', '.ppt']:
                code_extractions = await self.extract_code_from_powerpoint(file_path)

        except ImportError:
            print(f"Office document parsing libraries not available for {file_path}")

        return code_extractions

    async def extract_code_from_text(self, text, file_path, extraction_method):
        """Extract code patterns from any text content"""

        code_extractions = []

        # Pattern 1: Function definitions
        function_patterns = [
            r'def\s+(\w+)\s*\([^)]*\):', # Python functions
            r'function\s+(\w+)\s*\([^)]*\)\s*{', # JavaScript functions
            r'(\w+)\s*=\s*\([^)]*\)\s*=>', # Arrow functions
            r'class\s+(\w+)', # Class definitions
            r'interface\s+(\w+)', # TypeScript interfaces
            r'type\s+(\w+)\s*=', # Type definitions
        ]

        for pattern in function_patterns:
            matches = re.finditer(pattern, text, re.MULTILINE)
```

```
    for match in matches:
        # Extract surrounding context
        start = max(0, match.start() - 100)
        end = min(len(text), match.end() + 200)
        context = text[start:end]

        code_extractions.append({
            "type": "function_definition",
            "language": self.detect_language_from_pattern(pattern),
            "code": context,
            "function_name": match.group(1),
            "file_path": file_path,
            "extraction_method": extraction_method
        })

# Pattern 2: Code blocks (indented consistently)
code_block_pattern = r'(?:^    .+$\n?)+'
code_blocks = re.findall(code_block_pattern, text, re.MULTILINE)
for code_block in code_blocks:
    if len(code_block.strip()) > 50:  # Only substantial blocks
        code_extractions.append({
            "type": "indented_code_block",
            "language": "unknown",
            "code": code_block.strip(),
            "file_path": file_path,
            "extraction_method": extraction_method
        })

# Pattern 3: Import/require statements
import_patterns = [
    r'import\s+.*?from\s+[\'"][^\'"]+[\'"]',
    r'require\s*\(\s*[\'"][^\'"]+[\'"]',
    r'from\s+[\w.]+\s+import\s+.*',
    r'#include\s*<[^>]+>',
    r'using\s+[\w.]+;'
]

for pattern in import_patterns:
    matches = re.findall(pattern, text)
    for match in matches:
        code_extractions.append({
            "type": "import_statement",
            "language": self.detect_language_from_import(match),
            "code": match,
            "file_path": file_path,
            "extraction_method": extraction_method
        })
```

```
        return code_extractions
```

## Comprehensive Component Analysis

### Component Discovery Engine

python

```python
class ComponentDiscoveryEngine:
    """Analyzes extracted code to identify all components, features, and functionality"""

    def __init__(self):
        self.analyzers = {
            "python": PythonComponentAnalyzer(),
            "javascript": JavaScriptComponentAnalyzer(),
            "typescript": TypeScriptComponentAnalyzer(),
            "html": HTMLComponentAnalyzer(),
            "css": CSSComponentAnalyzer(),
            "sql": SQLComponentAnalyzer(),
            "shell": ShellComponentAnalyzer(),
            "docker": DockerComponentAnalyzer(),
            "yaml": YAMLComponentAnalyzer(),
            "unknown": GenericComponentAnalyzer()
        }

    async def analyze_comprehensive_components(self, code_extractions):
        """Analyze all extracted code to identify components and features"""

        comprehensive_analysis = {
            "functions": [],
            "classes": [],
            "variables": [],
            "imports": [],
            "apis": [],
            "configurations": [],
            "database_schemas": [],
            "ui_components": [],
            "workflows": [],
            "features": [],
            "utilities": [],
            "constants": [],
            "types": [],
            "interfaces": []
        }

        for extraction in code_extractions:
            language = extraction.get("language", "unknown")
            analyzer = self.analyzers.get(language, self.analyzers["unknown"])

            analysis_result = await analyzer.analyze_code_components(extraction)

            # Merge results into comprehensive analysis
            for component_type, components in analysis_result.items():
                if component_type in comprehensive_analysis:
```

```
                comprehensive_analysis[component_type].extend(components)

    # Cross-reference and deduplicate
    comprehensive_analysis = await self.cross_reference_components(comprehensive_analysis)

    return comprehensive_analysis

async def identify_lost_functionality(self, comprehensive_analysis, project_root):
    """Identify valuable functionality that exists but isn't properly referenced"""

    lost_functionality = []

    # Find functions that are defined but never called
    unreferenced_functions = await self.find_unreferenced_functions(
        comprehensive_analysis["functions"], project_root
    )

    # Find classes that are defined but never instantiated
    unreferenced_classes = await self.find_unreferenced_classes(
        comprehensive_analysis["classes"], project_root
    )

    # Find utility functions in documentation that aren't in main codebase
    doc_only_utilities = await self.find_documentation_only_utilities(
        comprehensive_analysis["utilities"], project_root
    )

    # Find configuration options that exist but aren't documented
    undocumented_configs = await self.find_undocumented_configurations(
        comprehensive_analysis["configurations"], project_root
    )

    # Find incomplete feature implementations
    incomplete_features = await self.find_incomplete_feature_implementations(
        comprehensive_analysis["features"], project_root
    )

    lost_functionality.extend([
        {"type": "unreferenced_functions", "items": unreferenced_functions},
        {"type": "unreferenced_classes", "items": unreferenced_classes},
        {"type": "doc_only_utilities", "items": doc_only_utilities},
        {"type": "undocumented_configs", "items": undocumented_configs},
        {"type": "incomplete_features", "items": incomplete_features}
    ])

    return lost_functionality
```

```python
class PythonComponentAnalyzer:
    """Specialized analyzer for Python code components"""

    async def analyze_code_components(self, extraction):
        """Comprehensive analysis of Python code components"""

        code = extraction["code"]
        components = {
            "functions": [],
            "classes": [],
            "variables": [],
            "imports": [],
            "apis": [],
            "configurations": [],
            "utilities": [],
            "constants": [],
            "types": []
        }

        try:
            # Parse AST for comprehensive analysis
            tree = ast.parse(code)

            for node in ast.walk(tree):
                if isinstance(node, ast.FunctionDef):
                    function_analysis = await self.analyze_function(node, extraction)
                    components["functions"].append(function_analysis)

                elif isinstance(node, ast.ClassDef):
                    class_analysis = await self.analyze_class(node, extraction)
                    components["classes"].append(class_analysis)

                elif isinstance(node, ast.Assign):
                    variable_analysis = await self.analyze_assignment(node, extraction)
                    components["variables"].extend(variable_analysis)

                elif isinstance(node, ast.Import) or isinstance(node, ast.ImportFrom):
                    import_analysis = await self.analyze_import(node, extraction)
                    components["imports"].append(import_analysis)

                elif isinstance(node, ast.Constant) and isinstance(node.value, str):
                    if self.looks_like_configuration(node.value):
                        config_analysis = await self.analyze_configuration(node, extraction)
                        components["configurations"].append(config_analysis)

        except SyntaxError:
            # If AST parsing fails, use regex-based analysis
```

```
            components = await self.analyze_python_with_regex(code, extraction)

        return components

    async def analyze_function(self, node, extraction):
        """Detailed analysis of a Python function"""

        # Extract function signature
        args = [arg.arg for arg in node.args.args]

        # Analyze function body for patterns
        complexity = len(list(ast.walk(node)))
        has_docstring = (isinstance(node.body[0], ast.Expr) and
                         isinstance(node.body[0].value, ast.Constant))

        # Detect function purpose patterns
        purpose = await self.detect_function_purpose(node)

        # Extract decorators
        decorators = [ast.unparse(decorator) for decorator in node.decorator_list]

        return {
            "name": node.name,
            "arguments": args,
            "line_number": node.lineno,
            "complexity": complexity,
            "has_docstring": has_docstring,
            "decorators": decorators,
            "purpose": purpose,
            "is_async": isinstance(node, ast.AsyncFunctionDef),
            "file_path": extraction["file_path"],
            "extraction_context": extraction["type"]
        }

    async def detect_function_purpose(self, node):
        """Detect the purpose/category of a function based on patterns"""

        function_body = ast.unparse(node)

        if re.search(r'@app\.|@router\.|@api\.', function_body):
            return "api_endpoint"
        elif re.search(r'@pytest\.|assert|test_', node.name):
            return "test_function"
        elif re.search(r'async\s+def.*await', function_body):
            return "async_operation"
        elif re.search(r'return.*render|template|html', function_body):
            return "view_function"
```

```
        elif re.search(r'validate|check|verify', node.name):
            return "validation_function"
        elif re.search(r'parse|extract|transform', node.name):
            return "data_processing"
        elif re.search(r'save|create|update|delete', node.name):
            return "crud_operation"
        elif re.search(r'send|post|get|fetch', node.name):
            return "network_operation"
        elif re.search(r'log|debug|error|info', node.name):
            return "logging_function"
        else:
            return "general_utility"

class JavaScriptComponentAnalyzer:
    """Specialized analyzer for JavaScript/TypeScript code components"""

    async def analyze_code_components(self, extraction):
        """Comprehensive analysis of JavaScript/TypeScript components"""

        code = extraction["code"]
        components = {
            "functions": [],
            "classes": [],
            "variables": [],
            "imports": [],
            "ui_components": [],
            "apis": [],
            "utilities": [],
            "constants": [],
            "types": [],
            "interfaces": []
        }

        # Function patterns
        function_patterns = [
            r'function\s+(\w+)\s*\([^)]*\)\s*{',  # Regular functions
            r'const\s+(\w+)\s*=\s*\([^)]*\)\s*=>', # Arrow functions
            r'(\w+)\s*:\s*\([^)]*\)\s*=>', # Object method arrow functions
            r'async\s+function\s+(\w+)', # Async functions
            r'export\s+function\s+(\w+)', # Exported functions
        ]

        for pattern in function_patterns:
            matches = re.finditer(pattern, code)
            for match in matches:
                function_analysis = await self.analyze_js_function(match, code, extraction)
                components["functions"].append(function_analysis)
```

```
        # React component patterns
        react_patterns = [
            r'const\s+(\w+)\s*=\s*\(\)\s*=>\s*{',  # Functional components
            r'function\s+(\w+)\s*\([^)]*\)\s*{.*?return.*?<', # Function components
            r'class\s+(\w+)\s+extends\s+React\.Component', # Class components
        ]

        for pattern in react_patterns:
            matches = re.finditer(pattern, code, re.DOTALL)
            for match in matches:
                component_analysis = await self.analyze_react_component(match, code, extraction)
                components["ui_components"].append(component_analysis)

        # TypeScript interface patterns
        interface_pattern = r'interface\s+(\w+)\s*{([^}]+)}'
        interface_matches = re.finditer(interface_pattern, code)
        for match in interface_matches:
            interface_analysis = await self.analyze_typescript_interface(match, extraction)
            components["interfaces"].append(interface_analysis)

        return components
```

## Comprehensive Indexing System

### Feature Index Manager

python

```python
class FeatureIndexManager:
    """Creates and maintains comprehensive indexes of all discovered functionality"""

    def __init__(self, project_root):
        self.project_root = project_root
        self.index_file = f"{project_root}/.agent_zero/feature_index.json"
        self.cross_reference_file = f"{project_root}/.agent_zero/cross_references.json"
        self.lost_features_file = f"{project_root}/.agent_zero/lost_features.json"

    async def create_comprehensive_index(self, comprehensive_analysis):
        """Create master index of all discovered functionality"""

        master_index = {
            "metadata": {
                "created": datetime.now().isoformat(),
                "project_root": self.project_root,
                "total_files_analyzed": len(set(item.get("file_path", "")
                                        for category in comprehensive_analysis.values()
                                        for item in category)),
                "analysis_version": "1.0.0"
            },
            "categories": {},
            "cross_references": {},
            "search_indexes": {},
            "feature_map": {},
            "dependency_graph": {},
            "usage_tracking": {}
        }

        # Build category indexes
        for category, items in comprehensive_analysis.items():
            master_index["categories"][category] = await self.build_category_index(category, items)

        # Build cross-references
        master_index["cross_references"] = await self.build_cross_references(comprehensive_analysis)

        # Build search indexes
        master_index["search_indexes"] = await self.build_search_indexes(comprehensive_analysis)

        # Build feature map
        master_index["feature_map"] = await self.build_feature_map(comprehensive_analysis)

        # Build dependency graph
        master_index["dependency_graph"] = await self.build_dependency_graph(comprehensive_analysis)

        # Save master index
```

```
    await self.save_index(master_index)

    return master_index

async def build_category_index(self, category, items):
    """Build detailed index for specific category"""

    category_index = {
        "total_count": len(items),
        "items": {},
        "by_file": {},
        "by_purpose": {},
        "by_complexity": {},
        "orphaned": [],
        "frequently_used": [],
        "duplicates": []
    }

    for item in items:
        item_id = f"{category}_{item.get('name', 'unknown')}_{hash(str(item))}"

        # Main item entry
        category_index["items"][item_id] = {
            **item,
            "item_id": item_id,
            "category": category,
            "indexed_at": datetime.now().isoformat()
        }

        # Group by file
        file_path = item.get("file_path", "unknown")
        if file_path not in category_index["by_file"]:
            category_index["by_file"][file_path] = []
        category_index["by_file"][file_path].append(item_id)

        # Group by purpose
        purpose = item.get("purpose", "unknown")
        if purpose not in category_index["by_purpose"]:
            category_index["by_purpose"][purpose] = []
        category_index["by_purpose"][purpose].append(item_id)

    return category_index

async def build_search_indexes(self, comprehensive_analysis):
    """Build multiple search indexes for fast lookup"""

    search_indexes = {
```

```
        "by_name": {},
        "by_keyword": {},
        "by_file_path": {},
        "by_signature": {},
        "by_purpose": {},
        "full_text": {}
    }

    for category, items in comprehensive_analysis.items():
        for item in items:
            item_id = f"{category}_{item.get('name', 'unknown')}_{hash(str(item))}"

            # Name index
            name = item.get("name", "")
            if name:
                if name not in search_indexes["by_name"]:
                    search_indexes["by_name"][name] = []
                search_indexes["by_name"][name].append(item_id)

            # Keyword index (extract from code and names)
            keywords = await self.extract_keywords_from_item(item)
            for keyword in keywords:
                if keyword not in search_indexes["by_keyword"]:
                    search_indexes["by_keyword"][keyword] = []
                search_indexes["by_keyword"][keyword].append(item_id)

            # File path index
            file_path = item.get("file_path", "")
            if file_path:
                if file_path not in search_indexes["by_file_path"]:
                    search_indexes["by_file_path"][file_path] = []
                search_indexes["by_file_path"][file_path].append(item_id)

    return search_indexes

async def identify_lost_and_duplicate_features(self, master_index):
    """Identify lost functionality and duplicates"""

    lost_features = {
        "unreferenced_valuable": [],
        "documentation_only": [],
        "incomplete_implementations": [],
        "outdated_references": [],
        "potential_duplicates": []
    }

    # Find unreferenced valuable functionality
```

```
    for category, category_data in master_index["categories"].items():
        for item_id, item in category_data["items"].items():
            if await self.is_valuable_but_unreferenced(item, master_index):
                lost_features["unreferenced_valuable"].append({
                    "item_id": item_id,
                    "item": item,
                    "reason": "Valuable functionality exists but no references found"
                })

    # Find documentation-only features
    doc_features = await self.find_documentation_only_features(master_index)
    lost_features["documentation_only"].extend(doc_features)

    # Find potential duplicates
    duplicates = await self.find_duplicate_functionality(master_index)
    lost_features["potential_duplicates"].extend(duplicates)

    # Save lost features analysis
    await self.save_lost_features_analysis(lost_features)

    return lost_features

async def is_valuable_but_unreferenced(self, item, master_index):
    """Determine if an item is valuable but unreferenced"""

    # Check if it's a substantial implementation
    if item.get("complexity", 0) < 10:  # Skip trivial items
        return False

    # Check if it has documentation (suggests intentional development)
    if item.get("has_docstring") or item.get("has_comments"):
        # Look for any references in cross-reference graph
        item_id = item.get("item_id")
        references = master_index["cross_references"].get(item_id, [])

        if len(references) == 0:
            return True

    return False

async def generate_feature_recovery_recommendations(self, lost_features):
    """Generate actionable recommendations for recovering lost functionality"""

    recommendations = []

    for category, items in lost_features.items():
        for item in items:
```

```
        if category == "unreferenced_valuable":
            recommendations.append({
                "type": "expose_valuable_function",
                "item": item,
                "action": "Add to main API or create wrapper function",
                "effort": "Low",
                "value": "High"
            })

        elif category == "documentation_only":
            recommendations.append({
                "type": "implement_documented_feature",
                "item": item,
                "action": "Implement the functionality described in documentation",
                "effort": "Medium",
                "value": "High"
            })

        elif category == "potential_duplicates":
            recommendations.append({
                "type": "consolidate_duplicates",
                "item": item,
                "action": "Review and consolidate duplicate implementations",
                "effort": "Medium",
                "value": "Medium"
            })

    return recommendations
```

---

## 🔗 Integration with Agent Zero Ecosystem

### ERDU Integration for Feature Discovery

yaml

```
Feature_Discovery_ERDU_Enhancement:
  Loop_1_Evaluate:
    - "Continuous monitoring for new code additions and feature development"
    - "Detection of orphaned functionality and unreferenced valuable code"
    - "Analysis of feature usage patterns and identification of declining features"

  Loop_2_Research:
    - "Comprehensive code archaeology across all file formats"
    - "Cross-reference analysis to understand feature relationships"
    - "Documentation analysis to identify described but unimplemented features"

  Loop_3_Decide:
    - "Prioritization of lost feature recovery based on value and effort"
    - "Decision framework for duplicate consolidation vs. preservation"
    - "Strategic planning for feature exposure and documentation"

  Loop_4_Utilize:
    - "Automated feature recovery implementation"
    - "Integration of discovered functionality into main codebase"
    - "Creation of proper cross-references and documentation"

  Loop_5_Optimize:
    - "Feature usage tracking and optimization recommendations"
    - "Continuous improvement of feature discovery algorithms"
    - "Enhancement of indexing and cross-referencing capabilities"
```

## Template Integration

yaml

```
Feature_Discovery_Templates:
  feature_recovery_workflow:
    - "Parse project comprehensively across all formats"
    - "Identify lost and unreferenced functionality"
    - "Generate recovery recommendations with effort estimates"
    - "Implement feature exposure and integration"

  code_archaeology_audit:
    - "Complete project code inventory"
    - "Cross-reference mapping and dependency analysis"
    - "Duplicate detection and consolidation recommendations"
    - "Documentation alignment with actual codebase"

  feature_optimization_workflow:
    - "Usage pattern analysis and optimization opportunities"
    - "Dead code identification vs. valuable unreferenced code"
    - "Feature consolidation and API improvement suggestions"
    - "Documentation generation for discovered features"
```

## Integration with AZ300 Debug Agent

python

```python
class DebugArchaeologyIntegration:
    """Integration between AZ400 Code Archaeologist and AZ300 Debug Agent"""

    async def provide_context_for_debugging(self, error_context):
        """Provide comprehensive code context for debugging"""

        # Find all related functionality
        related_features = await self.find_features_related_to_error(error_context)

        # Check if error involves unreferenced functionality
        unreferenced_solutions = await self.find_unreferenced_solutions(error_context)

        # Identify potential duplicate implementations that might work
        alternative_implementations = await self.find_alternative_implementations(error_context)

        return {
            "related_features": related_features,
            "unreferenced_solutions": unreferenced_solutions,
            "alternative_implementations": alternative_implementations,
            "comprehensive_context": await self.build_comprehensive_error_context(error_context)
        }
```

# 🚀 Implementation & Deployment

## Immediate Deployment Strategy

```bash
# Phase 1: Basic code extraction (Day 1-2)
python deploy_code_archaeologist.py --basic-extraction
# Result: Parse all .py, .js, .md files and create basic inventory

# Phase 2: Advanced parsing (Day 3-4)
python deploy_advanced_parsing.py --pdf-extraction --office-docs
# Result: Extract code from PDFs, Word docs, PowerPoint

# Phase 3: Comprehensive indexing (Day 5-7)
python deploy_comprehensive_indexing.py --cross-references --lost-features
# Result: Complete feature map with lost functionality identification
```

## Integration with Existing Systems

```python
# Add to existing Agent Zero workflow
@app.post("/archaeology/discover")
async def discover_lost_features():
    archaeologist = CodeArchaeologist()
    results = await archaeologist.comprehensive_project_analysis()
    return results

# Add to existing template workflows
WORKFLOW_CHAINS.update({
    "comprehensive_feature_audit": [
        {"template": "code_archaeology_scan", "agent": "AZ400-Archaeologist"},
        {"template": "feature_recovery_planning", "agent": "AZ400-Archaeologist"},
        {"template": "implementation_prioritization", "agent": "AZ100-Memory"},
        {"template": "documentation_alignment", "agent": "AZ115-Archivist"}
    ]
})
```

## Success Metrics

- **100% file format coverage**: Parse code from any format where it might exist
- **Lost feature discovery**: Identify 80%+ of unreferenced valuable functionality
- **Feature recovery**: Enable recovery of 90%+ of identified lost features
- **Comprehensive indexing**: Create searchable index of 100% of project functionality

---

## 🎯 Bottom Line: No More Lost Functionality

**AZ400 Code Archaeologist solves the critical problem of "lost" functionality by:**

✅ **Comprehensive parsing**: Extract code from ANY file format (PDFs, docs, markdown, etc.) ✅ **Deep analysis**: Identify functions, classes, features, utilities, configurations ✅ **Cross-referencing**: Map all relationships and dependencies
✅ **Lost feature recovery**: Find valuable but unreferenced functionality ✅ **Comprehensive indexing**: Searchable inventory of ALL project capabilities ✅ **Continuous monitoring**: Track new features and prevent future "loss"

**Ready to deploy and recover all your lost functionality?**