**Project Citadel: Detailed Technical Specifications**

**Core Modules**

**1. insert\_docs.py - Main Ingestion Orchestrator**

**Purpose**

Command-line utility for intelligent document crawling, content processing, and ChromaDB ingestion with automatic URL type detection and adaptive processing strategies.

**Functional Specification**

**Primary Capabilities:**

* **Smart URL Detection**: Automatically identifies URL type (regular website, .txt/markdown file, or sitemap)
* **Multi-Strategy Crawling**: Applies appropriate crawling method based on content type
* **Hierarchical Content Chunking**: Splits content by header structure for optimal retrieval
* **Metadata Extraction**: Captures document structure and source information
* **Batch Database Operations**: Efficient ChromaDB insertion with configurable batch sizes

**Input Parameters:**

bash

python insert\_docs.py <URL> [options]

--collection: ChromaDB collection name (default: "docs")

--db-dir: Database storage directory (default: "./chroma\_db")

--embedding-model: Embedding model (default: "all-MiniLM-L6-v2")

--chunk-size: Maximum chunk size in characters (default: 1000)

--max-depth: Recursion depth for regular URLs (default: 3)

--max-concurrent: Maximum parallel sessions (default: 10)

--batch-size: ChromaDB insertion batch size (default: 100)

**Technical Specification**

**Core Functions:**

1. **smart\_chunk\_markdown(markdown: str, max\_len: int = 1000) -> List[str]**
   * **Purpose**: Hierarchical content splitting strategy
   * **Algorithm**:
     + Level 1: Split by # headers
     + Level 2: Split by ## headers if chunks exceed max\_len
     + Level 3: Split by ### headers if still too large
     + Level 4: Character-based splitting as fallback
   * **Output**: List of optimally-sized text chunks
2. **URL Type Detection Functions:**
   * **is\_sitemap(url: str) -> bool**: Detects XML sitemaps
   * **is\_txt(url: str) -> bool**: Identifies text/markdown files
3. **Crawling Strategy Functions:**
   * **crawl\_recursive\_internal\_links()**: For regular websites
   * **crawl\_markdown\_file()**: For direct text files
   * **crawl\_batch()**: For sitemap URL lists
4. **extract\_section\_info(chunk: str) -> Dict[str, Any]**
   * **Extracts**: Headers, character count, word count
   * **Format**: Structured metadata for vector storage

**Data Flow:**

css

URL Input → Type Detection → Strategy Selection → Content Crawling

↓

Markdown Conversion → Hierarchical Chunking → Metadata Extraction

↓

Vector Embedding → ChromaDB Insertion → Success Confirmation

**Dependencies:**

* crawl4ai: Web crawling framework
* chromadb: Vector database operations
* utils.py: Database utilities
* requests: HTTP operations
* xml.etree.ElementTree: XML parsing

**2. rag\_agent.py - AI Agent with Retrieval Capabilities**

**Purpose**

Pydantic AI-powered agent that combines semantic document retrieval with LLM reasoning for intelligent question answering.

**Functional Specification**

**Core Capabilities:**

* **Semantic Document Retrieval**: Vector similarity search across ChromaDB collections
* **Tool-Calling Architecture**: Pydantic AI agent with retrieval tool integration
* **Context-Aware Response Generation**: Combines retrieved context with LLM inference
* **Flexible Model Support**: Configurable OpenAI model selection

**Usage Interface:**

bash

python rag\_agent.py --question "Your question here" [options]

--collection: ChromaDB collection name

--db-dir: Database directory path

--embedding-model: Embedding model name

--n-results: Number of retrieval results

**Technical Specification**

**Architecture Components:**

1. **RAGDeps Class**

python

@dataclass

class RAGDeps:

chroma\_client: chromadb.PersistentClient

collection\_name: str

embedding\_model: str

* + **Purpose**: Dependency injection for agent context
  + **Components**: Database client, collection reference, model configuration

1. **retrieve Tool Function**

python

@agent.tool

async def retrieve(context: RunContext[RAGDeps],

search\_query: str,

n\_results: int = 5) -> str

* + **Purpose**: Semantic document retrieval
  + **Process**: Query embedding → Vector search → Context formatting
  + **Output**: Formatted context string with relevance scores

1. **Agent Configuration**

python

agent = Agent(

model=os.getenv("MODEL\_CHOICE", "gpt-4.1-mini"),

deps\_type=RAGDeps,

system\_prompt="Context-aware assistant prompt..."

)

**Processing Pipeline:**

markdown

User Question → Agent Analysis → Tool Selection → Vector Retrieval

↓

Context Formatting → LLM Prompt Construction → Response Generation

↓

Source Attribution → Response Delivery

**Key Features:**

* **Environment-Based Configuration**: .env file integration
* **Error Handling**: API key validation and fallback responses
* **Async Operations**: Non-blocking I/O for scalability
* **Modular Design**: Configurable components for different use cases

**3. utils.py - Shared ChromaDB Utilities**

**Purpose**

Centralized utility functions for ChromaDB operations, providing consistent database management across the application.

**Functional Specification**

**Core Utilities:**

* **Database Client Management**: Persistent client creation with directory handling
* **Collection Operations**: Create/retrieve collections with embedding configuration
* **Document Management**: Batch insertion and query operations
* **Result Formatting**: Context preparation for AI agent consumption

**Technical Specification**

**Function Specifications:**

1. **get\_chroma\_client(persist\_directory: str) -> chromadb.PersistentClient**
   * **Purpose**: Database client initialization
   * **Features**: Directory creation, persistent storage configuration
   * **Return**: Configured ChromaDB client
2. **get\_or\_create\_collection()**

python

def get\_or\_create\_collection(

client: chromadb.PersistentClient,

collection\_name: str,

embedding\_model\_name: str = "all-MiniLM-L6-v2",

distance\_function: str = "cosine"

) -> chromadb.Collection

* + **Features**: SentenceTransformer embedding function, cosine similarity
  + **Error Handling**: Graceful collection creation if not exists

1. **add\_documents\_to\_collection()**

python

def add\_documents\_to\_collection(

collection: chromadb.Collection,

ids: List[str],

documents: List[str],

metadatas: Optional[List[Dict[str, Any]]] = None,

batch\_size: int = 100

) -> None

* + **Features**: Batch processing, memory efficiency, metadata handling
  + **Algorithm**: more\_itertools.batched for optimal performance

1. **query\_collection()**
   * **Parameters**: Query text, result count, optional filters
   * **Return**: Complete result set with documents, metadata, distances, IDs
2. **format\_results\_as\_context(query\_results: Dict[str, Any]) -> str**
   * **Purpose**: AI-readable context formatting
   * **Format**: Numbered documents with relevance scores and metadata
   * **Features**: Structured presentation for LLM consumption

**Design Principles:**

* **Reusability**: Common patterns abstracted into utilities
* **Error Resilience**: Graceful handling of edge cases
* **Performance**: Batch operations and memory efficiency
* **Flexibility**: Configurable parameters for different use cases

**Specialized Crawlers**

**4. 1-crawl\_single\_page.py - Single Page Extraction**

**Purpose**

Minimal example demonstrating basic single-page crawling functionality for testing and prototyping.

**Technical Specification**

**Core Function:**

python

async def main():

async with AsyncWebCrawler() as crawler:

result = await crawler.arun(url="https://ai.pydantic.dev/")

print(result.markdown)

**Features:**

* **Simple Interface**: Basic crawling with default configuration
* **Async Operation**: Non-blocking execution
* **Markdown Output**: Direct markdown content extraction

**Use Cases:**

* Quick content verification
* Development testing
* API exploration
* Content format validation

**5. 2-crawl\_docs\_sequential.py - Sequential Crawling with Session Reuse**

**Purpose**

Demonstrates efficient sequential crawling with browser session reuse for improved performance and resource management.

**Functional Specification**

**Key Features:**

* **Session Persistence**: Single browser session across multiple URLs
* **Sitemap Integration**: Automatic URL discovery from XML sitemaps
* **Resource Optimization**: Browser session reuse reduces overhead
* **Error Handling**: Per-URL success/failure tracking

**Technical Specification**

**Core Functions:**

1. **get\_pydantic\_ai\_docs\_urls() -> List[str]**

python

def get\_pydantic\_ai\_docs\_urls():

sitemap\_url = "https://ai.pydantic.dev/sitemap.xml"

# XML parsing with namespace handling

namespace = {'ns': 'http://www.sitemaps.org/schemas/sitemap/0.9'}

urls = [loc.text for loc in root.findall('.//ns:loc', namespace)]

* + **XML Parsing**: Robust namespace handling
  + **Error Recovery**: Graceful sitemap parsing failures

1. **crawl\_sequential(urls: List[str])**

python

browser\_config = BrowserConfig(

headless=True,

extra\_args=["--disable-gpu", "--disable-dev-shm-usage", "--no-sandbox"]

)

* + **Session Management**: Persistent session ID across requests
  + **Performance Options**: Docker/low-memory optimizations
  + **Sequential Processing**: Ordered URL processing with session reuse

**Processing Flow:**

markdown

Sitemap URL → XML Parsing → URL Extraction → Session Creation

↓

Sequential Crawling → Result Validation → Session Cleanup

**Performance Benefits:**

* **Reduced Overhead**: Browser session reuse
* **Memory Efficiency**: Single browser instance
* **Resource Control**: Controlled sequential processing

**6. 3-crawl\_sitemap\_in\_parallel.py - Parallel Sitemap Processing**

**Purpose**

High-performance parallel crawling system with memory management and resource monitoring for large-scale documentation processing.

**Functional Specification**

**Core Capabilities:**

* **Parallel Processing**: Concurrent URL crawling with configurable limits
* **Memory Monitoring**: Real-time memory usage tracking and reporting
* **Resource Management**: Adaptive dispatching based on system resources
* **Performance Metrics**: Success/failure statistics and memory profiling

**Technical Specification**

**Architecture Components:**

1. **Memory Monitoring System:**

python

peak\_memory = 0

process = psutil.Process(os.getpid())

def log\_memory(prefix: str = ""):

current\_mem = process.memory\_info().rss

# Peak memory tracking and reporting

1. **Adaptive Resource Management:**

python

dispatcher = MemoryAdaptiveDispatcher(

memory\_threshold\_percent=70.0,

check\_interval=1.0,

max\_session\_permit=max\_concurrent

)

* + **Memory Threshold**: 70% system memory limit
  + **Dynamic Monitoring**: 1-second check intervals
  + **Session Limits**: Configurable parallel session caps

1. **Parallel Crawling Engine:**

python

results = await crawler.arun\_many(

urls=urls,

config=crawl\_config,

dispatcher=dispatcher

)

**Performance Features:**

* **Batch Processing**: All URLs processed in parallel
* **Resource Awareness**: Memory-adaptive session management
* **Performance Monitoring**: Real-time memory and success tracking
* **Error Resilience**: Individual URL failure handling

**Monitoring Output:**

yaml

Current Memory: XXX MB, Peak: XXX MB

Successfully crawled: XX

Failed: XX

Peak memory usage (MB): XXX

**7. 4-crawl\_llms\_txt.py - Markdown/Text File Processing**

**Purpose**

Specialized processor for Markdown and text files with header-based content chunking for structured document processing.

**Functional Specification**

**Content Processing Strategy:**

* **Header Detection**: Regex-based # and ## header identification
* **Structural Chunking**: Content split by header boundaries
* **Content Validation**: Empty chunk filtering
* **Debugging Output**: Chunk enumeration and preview

**Technical Specification**

**Core Function:**

python

async def scrape\_and\_chunk\_markdown(url: str):

# Markdown content extraction

result = await crawler.arun(url=url, config=crawl\_config)

markdown = result.markdown

# Header-based chunking

header\_pattern = re.compile(r'^(# .+|## .+), re.MULTILINE)

headers = [m.start() for m in header\_pattern.finditer(markdown)]

headers.append(len(markdown))

# Chunk extraction

chunks = []

for i in range(len(headers)-1):

chunk = markdown[headers[i]:headers[i+1]].strip()

if chunk:

chunks.append(chunk)

**Processing Algorithm:**

1. **Pattern Matching**: Regex identification of # and ## headers
2. **Boundary Detection**: Header position indexing
3. **Content Segmentation**: Text extraction between boundaries
4. **Validation**: Empty content filtering
5. **Output**: Enumerated chunks with preview

**Use Cases:**

* **Documentation Processing**: Structured content extraction
* **Content Analysis**: Header-based organization
* **Chunk Validation**: Content structure verification
* **Development Testing**: Markdown processing validation

**8. 5-crawl\_site\_recursively.py - Recursive Site Crawling**

**Purpose**

Comprehensive recursive site crawler with depth control, URL deduplication, and memory-adaptive resource management for complete site documentation extraction.

**Functional Specification**

**Crawling Strategy:**

* **Depth-First Exploration**: Systematic site traversal with depth limits
* **URL Normalization**: Fragment removal for deduplication
* **Link Discovery**: Internal link extraction and queuing
* **Parallel Processing**: Batch crawling at each depth level

**Technical Specification**

**Core Algorithm:**

python

async def crawl\_recursive\_batch(start\_urls, max\_depth=3, max\_concurrent=10):

visited = set()

def normalize\_url(url):

return urldefrag(url)[0] # Remove fragments

current\_urls = set([normalize\_url(u) for u in start\_urls])

for depth in range(max\_depth):

urls\_to\_crawl = [normalize\_url(url) for url in current\_urls

if normalize\_url(url) not in visited]

if not urls\_to\_crawl:

break

# Parallel crawling at current depth

results = await crawler.arun\_many(

urls=urls\_to\_crawl,

config=run\_config,

dispatcher=dispatcher

)

# Link discovery for next depth

next\_level\_urls = set()

for result in results:

if result.success:

for link in result.links.get("internal", []):

next\_url = normalize\_url(link["href"])

if next\_url not in visited:

next\_level\_urls.add(next\_url)

current\_urls = next\_level\_urls

**Key Features:**

1. **URL Normalization:**
   * **Fragment Removal**: urldefrag() for consistent URLs
   * **Deduplication**: Set-based visited tracking
   * **Internal Link Filtering**: Domain-specific crawling
2. **Memory Management:**

python

dispatcher = MemoryAdaptiveDispatcher(

memory\_threshold\_percent=70.0,

check\_interval=1.0,

max\_session\_permit=max\_concurrent

)

1. **Depth Control:**
   * **Configurable Limits**: Maximum recursion depth
   * **Breadth-First Processing**: Complete depth level before proceeding
   * **Early Termination**: Stop when no new URLs found

**Processing Flow:**

sql

Start URLs → Normalization → Depth Level Processing → Link Discovery

↓

Parallel Crawling → Success Tracking → Next Level URL Collection

↓

Depth Increment → Termination Check → Results Compilation

**Performance Characteristics:**

* **Scalability**: Memory-adaptive resource management
* **Efficiency**: Parallel processing at each depth
* **Completeness**: Systematic site coverage
* **Resource Control**: Configurable concurrency limits

**Integration Architecture**

All specialized crawlers integrate seamlessly with the main insert\_docs.py orchestrator through:

**Common Interfaces:**

* Consistent return format: List[Dict[str, Any]] with url and markdown keys
* Standardized error handling and success validation
* Configurable parameters for performance tuning
* Memory-adaptive resource management

**Shared Dependencies:**

* crawl4ai: Core crawling framework
* asyncio: Asynchronous processing
* Resource management utilities
* Error handling patterns

This modular architecture enables flexible deployment strategies while maintaining consistent performance and reliability across different content types and scale requirements.