Project Citadel Technology Architecture

Executive Summary

Project Citadel is an advanced web crawling framework designed to efficiently extract, process, and store information from various web sources. Built primarily in Python, the system employs a modular architecture that supports multiple crawler implementations, each optimized for different crawling scenarios. The technology stack has been carefully selected to ensure scalability, maintainability, and extensibility, allowing the system to handle diverse web content while maintaining high performance and reliability.

This document outlines the core technology components of Project Citadel, their relationships and dependencies, version compatibility requirements, and provides recommendations for future technology decisions. It serves as a comprehensive reference for developers, system administrators, and stakeholders involved in the development, deployment, and maintenance of the Project Citadel system.

Core Technology Stack

Programming Languages

Language	Purpose	Key Libraries/Frameworks
Python 3.10+	Primary development language	FastAPI, Requests, BeautifulSoup4, Langchain, LangGraph
JavaScript	Client-side integration	React, CopilotKit
SQL	Database queries and management	SQLAlchemy, Alembic

Backend Framework

FastAPI serves as the primary backend framework for Project Citadel, providing:
- High-performance API endpoints with automatic OpenAPI documentation
- Asynchronous request handling for improved concurrency

- Type validation through Pydantic models
- Middleware support for authentication, logging, and error handling

Crawler Components

The core of Project Citadel consists of several specialized crawler implementations:

1. **Single Page Crawler**: Optimized for extracting content from individual web pages
   - Uses BeautifulSoup4 for HTML parsing
   - Implements custom content extraction algorithms
   - Supports various content types (text, images, metadata)

2. **Sequential Document Crawler**: Designed for navigating through multi-page documents
   - Maintains session state across page transitions
   - Implements breadcrumb tracking for navigation history
   - Supports pagination patterns detection

3. **Parallel Sitemap Crawler**: Processes websites with XML sitemaps
   - Parses sitemap.xml files to identify crawlable URLs
   - Implements concurrent crawling with configurable parallelism
   - Prioritizes URLs based on configurable rules

4. **Text/Markdown Crawler**: Specialized for processing plain text and markdown content
   - Implements markdown parsing and conversion
   - Extracts structured data from semi-structured text
   - Preserves document formatting and hierarchy

5. **Recursive Site Crawler**: For comprehensive website crawling
   - Implements depth-first and breadth-first crawling strategies
   - Respects robots.txt directives
   - Detects and avoids crawler traps

Common Utilities

Several utility components are shared across all crawler implementations:

1. **URL Validation and Normalization**:
   - URL parsing and validation using urllib
   - Relative to absolute URL conversion
   - URL deduplication and canonicalization

2. **HTML Parsing**:
   - BeautifulSoup4 for DOM manipulation

   - lxml for high-performance XML/HTML processing
   - Custom selectors for content extraction

3. **Rate Limiting**:
   - Token bucket algorithm implementation
   - Domain-specific rate limiting
   - Backoff strategies for handling 429 responses

4. **Error Handling**:
   - Comprehensive exception hierarchy
   - Retry mechanisms with exponential backoff
   - Detailed error logging and reporting

5. **HTTP Session Management**:
   - Cookie handling and session persistence
   - Header management and rotation
   - Proxy support for distributed crawling

Data Storage

1. **Vector Database**:
   - Qdrant for storing and querying vector embeddings
   - Supports semantic search capabilities
   - Implements efficient nearest neighbor search

2. **Relational Database**:
   - PostgreSQL via Supabase for structured data storage
   - SQLAlchemy ORM for database interactions
   - Alembic for schema migrations

3. **Caching Layer**:
   - Redis for high-performance caching
   - Implements TTL-based cache invalidation
   - Supports distributed caching across nodes

Integration Components

1. **LLM Orchestration**:
   - LangChain for LLM integration and workflow management
     - Custom prompt templates and chain management
     - Model selection and fallback strategies
     - Document processing and retrieval augmentation

    - LangGraph for complex, stateful AI workflows
    - Directed graph-based workflow orchestration
    - State management across multi-step LLM interactions
    - Cyclic and conditional execution paths
    - Seamless integration with LangChain components

2. **Protocol Handlers**:
   - AG-UI Protocol compliance adapters
    - Standardized interface for AI-GUI interactions
    - Structured message format for tool use and function calling
    - Cross-platform compatibility layer
    - Consistent error handling and response formatting

3. **CopilotKit Integration**:
   - UI component bridging for embedded AI assistance
   - Event handling and state management
   - Real-time collaboration features
   - Context-aware AI interactions within application interfaces
   - Customizable UI components for different assistance modes

4. **Ollama Integration Approaches**:
   - CLI-based Integration
    - Shell command execution for model management and inference
    - Subprocess handling for asynchronous operations
    - Simplified deployment with minimal dependencies
    - Limited to text completion workflows

• Direct REST API Integration
    - Native HTTP requests to Ollama API endpoints
    - Full access to model parameters and configurations
    - Streaming response support
    - Custom header management for authentication

• FastAPI Gateway Integration
    - Abstraction layer over Ollama API
    - Request validation and transformation

    - Response formatting and error handling
    - Caching and rate limiting capabilities
    - Unified interface for multiple model providers

- Supported Ollama Models:
    - deepcoder:14b (9.0 GB)
    - deepcoder:latest (9.0 GB) - same as deepcoder:14b
    - deepseek-r1:32b (19 GB)
    - deepseek-r1:latest (4.7 GB)
    - mistral:latest (4.1 GB)
    - deepcoder-bf16:latest (29 GB)

Technology Relationships and Dependencies

Component Dependency Graph

The Project Citadel system has the following key technology dependencies:

1. **Crawler Core Dependencies**:
   - Python 3.10+ runtime environment
   - Requests library for HTTP interactions
   - BeautifulSoup4 and lxml for HTML/XML parsing
   - urllib3 for URL handling and connection pooling

2. **API Layer Dependencies**:
   - FastAPI framework
   - Pydantic for data validation
   - Starlette for ASGI support
   - Uvicorn as the ASGI server

3. **Database Layer Dependencies**:
   - PostgreSQL 13+ for relational data
   - SQLAlchemy 2.0+ for ORM functionality
   - Alembic for database migrations
   - Qdrant for vector storage
   - Redis 6+ for caching

4. **Integration Layer Dependencies**:
   - LangChain for LLM orchestration and workflows

- LangGraph for complex, stateful AI agent workflows
- OpenAI API client for model access
- Ollama for local model deployment and inference
- CopilotKit for UI integration and embedded assistance
- Protocol adapters for AG-UI compliance

Critical Paths and Bottlenecks

The system architecture has several critical paths that require special attention:

1. **HTTP Request Handling**:
   - The Requests library is central to all crawler operations
   - Network latency and connection pooling affect overall performance
   - Proper timeout and retry configuration is essential

2. **HTML Parsing Efficiency**:
   - BeautifulSoup4 and lxml performance impacts content extraction speed
   - Large DOM trees can cause memory pressure
   - Selector optimization is critical for performance

3. **Database Interactions**:
   - Connection pooling configuration affects throughput
   - Query optimization is essential for large datasets
   - Transaction management impacts data consistency

4. **Vector Operations**:
   - Embedding generation is computationally intensive
   - Vector search performance depends on index configuration
   - Dimensionality and quantization affect storage requirements

5. **LLM Workflow Orchestration**:
   - LangChain and LangGraph execution efficiency
   - Token usage optimization for cost management
   - State persistence for complex workflows
   - Error handling and fallback strategies

Current Versions and Compatibility

Core Components

Component	Current Version	Minimum Version	Notes
Python	3.10.12	3.10.0	Type hints and async features require 3.10+
FastAPI	0.103.1	0.95.0	Dependency on Pydantic v2
Requests	2.31.0	2.28.0	HTTP/2 support recommended
BeautifulSoup4	4.12.2	4.10.0	HTML5 parser support required
SQLAlchemy	2.0.23	2.0.0	ORM features use 2.0 API
Pydantic	2.4.2	2.0.0	V2 API required for FastAPI integration
LangChain	0.0.335	0.0.300	Rapid development cycle, pin exact version
LangGraph	0.0.19	0.0.15	Requires compatible LangChain version
CopilotKit	0.14.0	0.12.0	React 18+ compatibility required
Redis-py	5.0.1	4.5.0	Cluster support required
Ollama	0.1.20	0.1.14	API stability improving with newer versions

External Services

Service	API Version	Authentication Method	Rate Limits
Qdrant	v1.6.1	API key	Depends on deployment
PostgreSQL	15.4	Username/password	Connection pool limits
OpenAI API	v1	API key	Tokens per minute, RPM
Supabase	v2	JWT	Depends on plan
Redis	7.2	Password	Memory limits
Ollama	v1	None (local) / API key (remote)	Hardware dependent

Compatibility Matrix

Component	Operating Systems	Container Support	Cloud Platforms
Crawler Core	Linux, macOS, Windows	Docker, Kubernetes	AWS, GCP, Azure
FastAPI Service	Linux, macOS, Windows	Docker, Kubernetes	AWS, GCP, Azure
Qdrant	Linux, macOS	Docker, Kubernetes	AWS, GCP, Azure
PostgreSQL	Linux, macOS, Windows	Docker, Kubernetes	AWS, GCP, Azure, Supabase
Redis	Linux, macOS	Docker, Kubernetes	AWS ElastiCache, GCP Memorystore, Azure Cache
LangChain/LangGraph	Linux, macOS, Windows	Docker, Kubernetes	AWS, GCP, Azure
Ollama	Linux, macOS, Windows (WSL)	Docker	Self-hosted, AWS, GCP, Azure
CopilotKit	Browser, React Native	N/A	Any hosting platform

Upgrade Considerations

When upgrading components, consider the following compatibility issues:

1. **Python Version Upgrades**:
   - Type hint syntax changes between Python versions
   - Async context manager behavior differences
   - Standard library API changes

2. **FastAPI/Pydantic Upgrades**:
   - Breaking changes between Pydantic v1 and v2
   - Middleware API changes in FastAPI
   - OpenAPI schema generation differences

3. **Database Upgrades**:
   - SQLAlchemy 2.0 API differs significantly from 1.x

   - PostgreSQL version affects available features
   - Migration testing required for schema changes

4. **External API Changes**:
   - OpenAI API versioning and model deprecation
   - Qdrant API evolution for vector operations
   - Supabase feature availability by version

5. **LLM Framework Upgrades**:
   - LangChain API changes between versions
   - LangGraph state management evolution
   - AG-UI Protocol specification updates
   - CopilotKit component API changes

6. **Ollama Integration Considerations**:
   - API endpoint changes between versions
   - Model format compatibility
   - Resource utilization differences
   - Authentication method changes

Deployment Architecture

Project Citadel is designed for deployment in containerized environments, with the following recommended configuration:

Kubernetes Deployment

1. **API Layer**:
   - FastAPI service deployed as stateless pods
   - Horizontal Pod Autoscaler for dynamic scaling
   - Ingress controller for traffic management

2. **Core Services**:
   - Crawler Core deployed as stateful sets
   - Agent Factory as deployment with replica sets
   - LLM Orchestrator as deployment with resource limits

3. **Crawler Agents**:
   - Each crawler type deployed as separate pods
   - Resource quotas based on crawler complexity
   - Affinity rules for optimal node placement

4. **LLM Integration Layer**:
   - LangChain/LangGraph orchestrator as deployment
   - Ollama instances as StatefulSet or DaemonSet
   - CopilotKit backend as deployment with replica sets

5. **Managed Services**:
   - Qdrant deployed as StatefulSet or managed service
   - PostgreSQL as managed service (Supabase or cloud provider)
   - Redis as StatefulSet with persistence or managed service

6. **Monitoring & Management**:
   - Prometheus for metrics collection
   - Grafana for visualization
   - Jaeger for distributed tracing

Resource Requirements

Component	CPU (requests/limits)	Memory (requests/limits)	Storage
FastAPI Service	0.5/1.0 CPU	512Mi/1Gi	N/A
Crawler Core	1.0/2.0 CPU	1Gi/2Gi	N/A
Single Page Crawler	0.2/0.5 CPU	256Mi/512Mi	N/A
Sequential Doc Crawler	0.3/0.6 CPU	384Mi/768Mi	N/A
Parallel Sitemap Crawler	0.5/1.0 CPU	512Mi/1Gi	N/A
Text/Markdown Crawler	0.2/0.4 CPU	256Mi/512Mi	N/A
Recursive Site Crawler	0.5/1.0 CPU	512Mi/1Gi	N/A
LangChain/LangGraph	0.5/1.0 CPU	1Gi/2Gi	N/A
Ollama (per model)	2.0/4.0 CPU	4Gi/8Gi	10Gi+
CopilotKit Backend	0.3/0.6 CPU	512Mi/1Gi	N/A
Qdrant	1.0/2.0 CPU	2Gi/4Gi	20Gi+
PostgreSQL	1.0/2.0 CPU	2Gi/4Gi	50Gi+
Redis	0.5/1.0 CPU	1Gi/2Gi	10Gi

Security Considerations

Authentication and Authorization

1. **API Security**:
   - JWT-based authentication
   - Role-based access control
   - API key management for service accounts

2. **Data Protection**:
   - Encryption at rest for sensitive data
   - TLS for all network communications
   - Proper secret management using Kubernetes secrets

3. **External Service Security**:
   - Secure API key rotation
   - IP allowlisting where supported
   - Minimal permission principle for service accounts

4. **LLM Integration Security**:
   - Prompt injection prevention
   - Input validation and sanitization
   - Output filtering for sensitive information
   - Rate limiting for API-based model access
   - Access control for local model deployments

Compliance Requirements

1. **Data Privacy**:
   - GDPR compliance for EU data
   - CCPA compliance for California residents
   - Data retention policies implementation

2. **Web Crawling Ethics**:
   - robots.txt compliance
   - Rate limiting to prevent site overload
   - User-agent identification

3. **AI Ethics and Governance**:
   - Transparency in AI-generated content
   - Bias monitoring and mitigation
   - Appropriate content filtering
   - User consent for AI interactions

Conclusion

Project Citadel's technology architecture provides a robust foundation for web crawling operations across diverse content types. The modular design allows for independent scaling of components based on workload requirements, while the standardized interfaces enable easy extension with new crawler implementations.

Key strengths of the architecture include:

1. **Modularity**: Clear separation of concerns between components
2. **Scalability**: Containerized deployment with horizontal scaling
3. **Extensibility**: Well-defined interfaces for adding new crawler types
4. **Reliability**: Comprehensive error handling and monitoring
5. **Performance**: Optimized components for high-throughput crawling
6. **AI Integration**: Flexible LLM orchestration with LangChain and LangGraph
7. **UI Capabilities**: Embedded assistance through CopilotKit and AG-UI Protocol

Future technology recommendations:

1. **Evaluate newer vector database options** as they emerge in the rapidly evolving space
2. **Consider streaming processing pipelines** for real-time content processing
3. **Explore serverless deployment options** for cost optimization of bursty workloads
4. **Implement circuit breakers** for improved resilience against external service failures
5. **Adopt GitOps practices** for infrastructure and deployment management
6. **Investigate fine-tuning options** for domain-specific LLM optimization
7. **Explore multi-modal capabilities** for processing diverse content types

By following the guidelines in this document, the Project Citadel system can be effectively developed, deployed, and maintained to meet the evolving requirements of web content extraction and processing.