SONA AI Assistant - Solution Design Document

Version: 1.0.0
Date: 26 May 2025
Document Type: Technical Architecture & Design Specification

---

1. Executive Summary

SONA (Smart Orchestrated Natural Assistant) is a modular AI-driven web assistant designed to provide intelligent conversational experiences through multiple interaction modalities. The system demonstrates enterprise-grade architecture with swappable AI components, comprehensive error handling, and production-ready deployment capabilities.

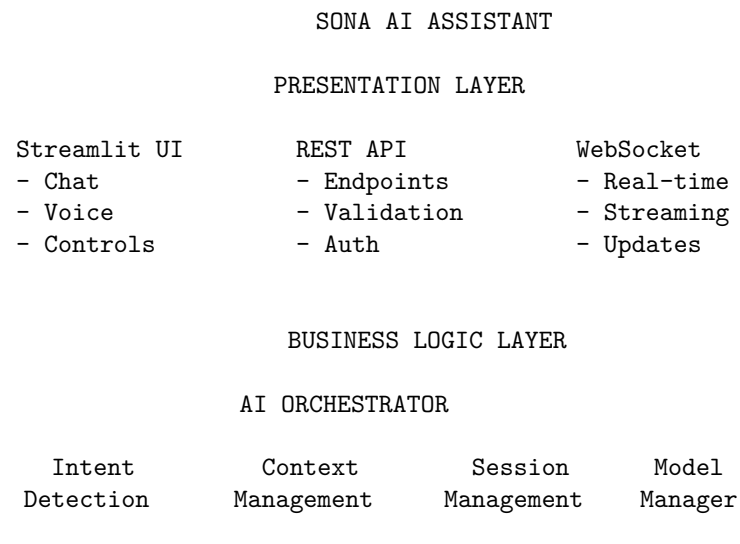
Key Objectives

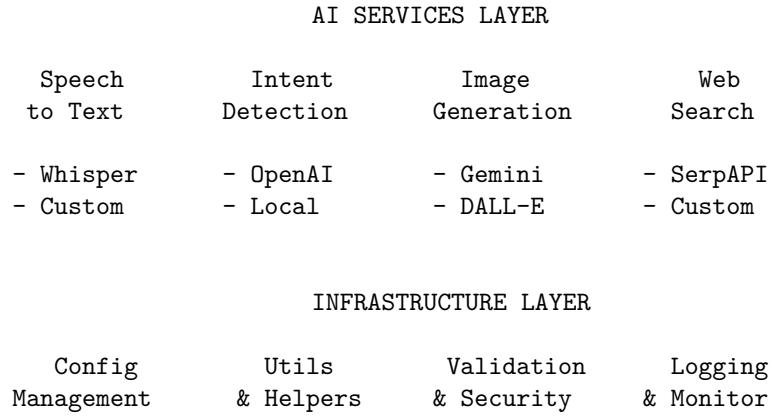
- **Multi-modal Interaction**: Support text, voice, and visual inputs
- **Modular Architecture**: Swappable AI services without code modifications
- **Production Readiness**: Containerized deployment with monitoring and scaling
- **Intelligent Orchestration**: Context-aware AI service coordination

---

2. High-Level Architecture

2.1 System Overview





2.2 Architectural Principles

1. **Modular Design**: Each AI service is independently swappable
2. **Loose Coupling**: Components communicate through well-defined interfaces
3. **Single Responsibility**: Each module has a clear, focused purpose
4. **Scalability**: Horizontal and vertical scaling support
5. **Fault Tolerance**: Graceful degradation and error recovery
6. **Observability**: Comprehensive logging, monitoring, and debugging

---

3. Technology Stack & Justification

3.1 Core Technologies

Component	Technology	Justification
Backend Framework	FastAPI	High performance, automatic API docs, async support, type hints
Frontend Framework	Streamlit	Rapid prototyping, Python-native, real-time updates
AI Orchestration	Custom Python	Full control, flexibility, easy integration

Component	Technology	Justification
Configuration	Pydantic Settings	Type-safe config, environment variable management
Logging	Loguru	Simple API, structured logging, performance
Containerization	Docker/Compose	Consistency, deployment simplicity, scaling
Validation	Pydantic	Runtime type checking, data validation

3.2 AI Service Providers

Service	Provider	Justification
Speech-to-Text	OpenAI Whisper API	High accuracy, multiple languages, robust API
Intent Detection	OpenAI GPT-3.5/4	Advanced reasoning, context understanding
Image Generation	Google Gemini	Visual description capabilities, API stability
Web Search	SerpAPI	Real-time data, comprehensive results, reliability

3.3 Development & Deployment

Aspect	Technology	Justification
Language	Python 3.11+	Rich AI ecosystem, rapid development, community
Package Management	pip + requirements.txt	Simplicity, compatibility, reproducibility
Environment Management	python-dotenv	Secure credential management, environment separation
Code Quality	black, ruff, flake8	Consistent formatting, linting, best practices

---

4. Detailed Module Breakdown

4.1 UI Module (`/ui`)

`ui/`

```
    __init__.py
    streamlit_app.py          # Main application entry
    components/
        __init__.py
        chat_interface.py     # Chat UI components
        voice_input.py        # Voice recording components
```

Responsibilities: - User interface rendering and interaction - Real-time audio recording and playback - Chat message display and formatting - File upload handling - Session state management

Key Components: - **SONAStreamlitApp**: Main application class - **ChatInterface**: Message rendering and formatting - **EnhancedVoiceInputComponent**: Audio capture and processing

Design Patterns: - Component-based architecture - Event-driven interactions - Reactive state management

4.2 Backend Module (`/backend`)

```
backend/
    __init__.py
    app.py                    # FastAPI application
    middleware/
        __init__.py
        logger.py            # Logging configuration
        error_handler.py     # Global error handling
```

Responsibilities: - REST API endpoint management - Request/response processing - Authentication and authorization - Error handling and logging - API documentation generation

Key Components: - **SONABackend**: Main FastAPI application - **ErrorHandlingMiddleware**: Global exception handling - **CORS Configuration**: Cross-origin request support

API Endpoints: - `GET /` - System information - `GET /health` - Health check with service status - `POST /api/v1/chat` - Text message processing - `POST /api/v1/upload-audio` - Audio file processing - `GET /api/v1/models` - Available AI models - `POST /api/v1/switch-model` - Runtime model switching

4.3 AI Orchestration Module (`/ai`)

```
ai/
    __init__.py
    orchestrator.py           # Main AI coordinator
    speech_to_text/
        __init__.py
        base.py              # Abstract base class
```

```
    whisper_service.py   # OpenAI Whisper implementation
    deepspeech_service.py # Future: Local DeepSpeech
  intent_detection/
    __init__.py
    base.py              # Abstract base class
    openai_service.py    # OpenAI GPT implementation
    local_transformer.py # Future: Local model
  image_generation/
    __init__.py
    base.py              # Abstract base class
    gemini_service.py    # Google Gemini implementation
    hybrid_service.py    # Gemini + DALL-E combination
  web_search/
     __init__.py
     base.py              # Abstract base class
     serp_service.py      # SerpAPI implementation
```

Responsibilities: - AI service coordination and routing - Model abstraction and switching - Context management and state - Service health monitoring - Error handling and fallbacks

Key Components: - **AIOrchestrator**: Central coordination hub - **Service Base Classes**: Abstract interfaces for each AI type - **Service Implementations**: Concrete AI provider integrations

Design Patterns: - Strategy Pattern: Swappable AI implementations - Factory Pattern: Service instantiation - Observer Pattern: Service health monitoring - Chain of Responsibility: Multi-step processing

4.4 Solana Wallet Module (`/wallet`) - Future Implementation

```
wallet/ (Planned)
  __init__.py
  wallet_manager.py        # Wallet operations
  transaction_handler.py   # Blockchain transactions
  solana_client.py        # Solana network interface
```

Planned Responsibilities: - Cryptocurrency wallet management - Solana blockchain interactions - Transaction processing and monitoring - Balance checking and reporting

Integration Points: - Intent detection for crypto queries - Web search for price information - Voice commands for wallet operations

4.5 Utility Modules (`/utils`)

```
utils/
  __init__.py
  constants.py             # System constants and enums
```

```
validation.py            # Input validation and sanitization
file_utils.py           # File operations and management
audio_utils.py          # Audio processing utilities
```

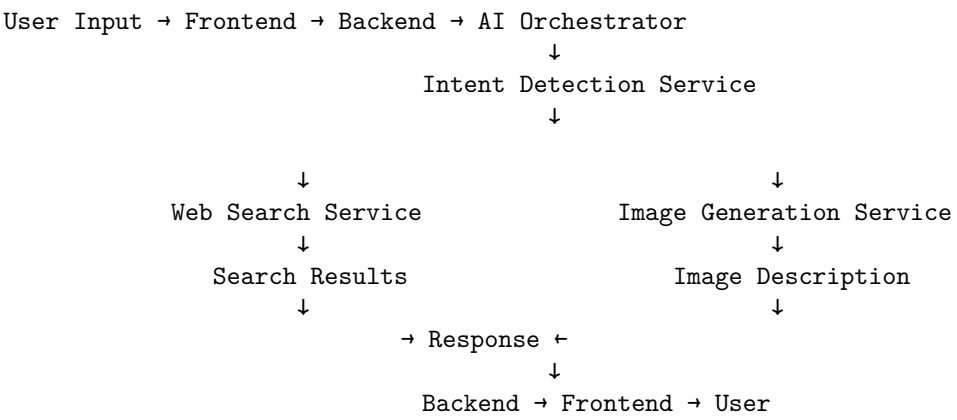
Responsibilities: - Input validation and sanitization - File handling and temporary storage - Audio processing and conversion - System constants and configuration - Security and data protection

Key Components: - **AudioProcessor**: Audio file validation and preprocessing - **Validation Functions**: Input sanitization and checking - **File Utilities**: Temporary file management and cleanup - **Constants**: Enums, error messages, configuration values

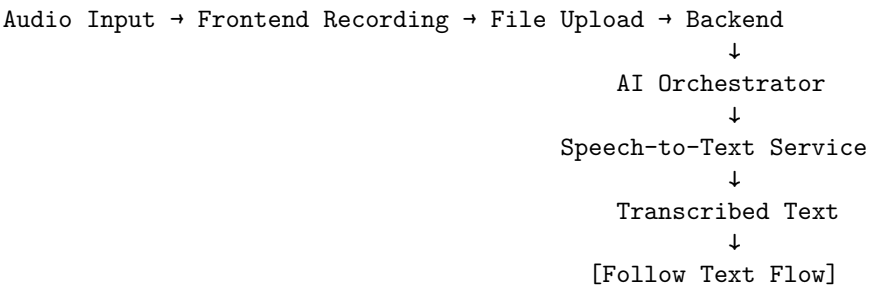
---

5. Application Flow Diagrams

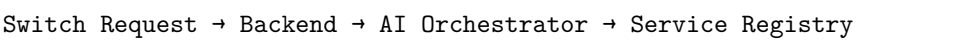
5.1 Text Chat Processing Flow



5.2 Voice Processing Flow



5.3 Model Switching Flow



```
                    ↓
Validate New Model
                    ↓
Initialize Service
                    ↓
Update Active Model
                    ↓
Return Success/Error
```

---

6. Prompt Handling and Response Logic

6.1 Intent Classification System

```python
# Intent Types
INTENTS = {
    "WEB_SEARCH": ["price", "current", "latest", "what is", "who is"],
    "IMAGE_GENERATION": ["image", "picture", "create", "generate", "draw"],
    "GENERAL_CHAT": ["hello", "how are you", "thank you"],
    "UNKNOWN": []  # Fallback
}
```

Processing Pipeline: 1. **Input Sanitization**: Remove harmful content, normalize text 2. **Context Analysis**: Consider conversation history and session state 3. **Intent Detection**: Use OpenAI GPT for sophisticated intent recognition 4. **Entity Extraction**: Identify key parameters (search terms, image prompts) 5. **Service Routing**: Direct to appropriate AI service 6. **Response Synthesis**: Combine results into coherent response

6.2 Response Generation Strategy

Search Results Processing: - Extract key information from multiple sources - Synthesize coherent answers using AI - Provide source attribution and links - Handle result quality and relevance

Image Generation Response: - Generate detailed visual descriptions - Create compact visual representations - Provide enhanced prompts and metadata - Handle generation failures gracefully

Error Handling: - Graceful degradation when services fail - User-friendly error messages - Automatic fallback to alternative services - Logging and monitoring for debugging

---

7. Session and State Management Plan

7.1 Session Architecture

```
# Session State Structure
SESSION_STATE = {
    "session_id": "unique_identifier",
    "user_context": {
        "conversation_history": [],
        "preferences": {},
        "active_models": {}
    },
    "ai_context": {
        "intent_history": [],
        "confidence_scores": [],
        "service_performance": {}
    },
    "system_state": {
        "service_health": {},
        "response_times": {},
        "error_counts": {}
    }
}
```

7.2 State Persistence Strategy

Frontend State (Streamlit): - Session-based state management - Automatic state persistence across interactions - Real-time UI updates and synchronization

Backend State: - In-memory session storage for development - Redis integration for production scaling - Database persistence for long-term storage

AI Service State: - Model configuration caching - Service health status tracking - Performance metrics collection

7.3 Context Management

Conversation Context: - Maintain message history and intent progression - Track user preferences and interaction patterns - Preserve context across model switches

AI Service Context: - Service availability and performance metrics - Model configuration and parameters - Error rates and fallback triggers

---

8. Deployment Strategy

8.1 Local Development

Setup Process: 1. Environment configuration with `.env` file 2. Python virtual environment creation 3. Dependency installation via `requirements.txt` 4. Service initialization and health checks 5. Development server startup

Development Features: - Hot reloading for rapid iteration - Comprehensive logging and debugging - Service mocking for offline development - Automated validation

8.2 Docker Containerization

Multi-Service Architecture:

```
services:
  sona-backend:
    - FastAPI application server
    - AI orchestrator and services
    - Health monitoring and logging

  sona-frontend:
    - Streamlit user interface
    - Real-time audio processing
    - Interactive chat components

  redis: (Optional)
    - Session state storage
    - Caching layer
    - Pub/sub messaging
```

Container Features: - Multi-stage builds for optimization - Health checks and monitoring - Volume mounts for persistent data - Environment variable configuration

8.3 AWS EC2 Cloud Deployment

Infrastructure Requirements: - **Instance Type**: t3.medium or larger (2 vCPU, 4GB RAM) - **Operating System**: Ubuntu 20.04 LTS - **Storage**: 20GB SSD minimum - **Security Groups**: HTTP (80), HTTPS (443), API (8000), UI (8501)

Deployment Process: 1. EC2 instance provisioning and configuration 2. Docker and Docker Compose installation 3. Application deployment and startup 4. SSL certificate configuration 5. Reverse proxy setup (Nginx) 6. Monitoring and alerting configuration

Production Considerations: - Load balancing for high availability - Auto-scaling group configuration - Database backup and recovery - Log aggregation

and monitoring - Security hardening and compliance - Performance optimization and caching

8.4 CI/CD Pipeline (Recommended)

Pipeline Stages: 1. **Source Control**: Git-based version control with feature branches 2. **Building**: Docker image creation and optimization 3. **Staging**: Deployment to staging environment for validation 4. **Production**: Blue-green deployment with rollback capability

Tools Integration: - **GitHub Actions** or **GitLab CI** for automation - **Docker Hub** or **AWS ECR** for image registry - **Terraform** for infrastructure as code - **Ansible** for configuration management

---

9. AI Model Abstraction and Switching Mechanism

9.1 Abstract Service Architecture

Base Class Design:

```python
# Abstract base class for all AI services
class AIServiceBase(ABC):
    def __init__(self, model_name: str, **config):
        self.model_name = model_name
        self.config = config
        self.is_initialized = False

    @abstractmethod
    async def initialize(self) -> None:
        """Service-specific initialization"""
        pass

    @abstractmethod
    async def process(self, input_data: Any) -> Dict[str, Any]:
        """Main processing method"""
        pass

    @abstractmethod
    def is_available(self) -> bool:
        """Service availability check"""
        pass

    async def health_check(self) -> Dict[str, Any]:
        """Health status reporting"""
        pass
```

9.2 Service Registry and Factory

Dynamic Service Loading:

```
# Service registry for dynamic model switching
SERVICE_REGISTRY = {
    "speech_to_text": {
        "whisper": WhisperService,
        "deepspeech": DeepSpeechService
    },
    "intent_detection": {
        "openai": OpenAIIntentService,
        "local_transformer": LocalTransformerService
    },
    "image_generation": {
        "gemini": GeminiImageService,
        "dalle": DalleImageService
    },
    "web_search": {
        "serp": SerpSearchService,
        "google": GoogleSearchService
    }
}
```

9.3 Runtime Model Switching

Switch Process: 1. **Validation**: Verify new model is available and supported 2. **Initialization**: Start new service with configuration 3. **Health Check**: Ensure service is operational 4. **Registry Update**: Update active model registry 5. **Cleanup**: Optionally cleanup previous service resources

Configuration Management: - Environment-based default models - User preference storage - Performance-based auto-switching

9.4 Fallback and Error Handling

Graceful Degradation: - Automatic fallback to backup services - Error rate monitoring and triggers - Service circuit breaker patterns - User notification of service changes

Performance Monitoring: - Response time tracking - Success/failure rate monitoring - Cost optimization based on usage - Quality metrics for different models

---

10. Security and Compliance

10.1 Data Security

Input Validation: - Comprehensive input sanitization - File type and size validation - Malicious content detection - Rate limiting and abuse prevention

API Security: - Request/response validation - Error message sanitization - Logging security events - API key rotation support

Data Protection: - Temporary file cleanup - Session data encryption - Secure credential storage - GDPR compliance considerations

10.2 Infrastructure Security

Container Security: - Non-root user execution - Minimal base images - Security scanning and updates - Resource limits and isolation

Network Security: - TLS/SSL encryption - Firewall configuration - Private network isolation - API endpoint protection

---

11. Performance and Scalability

11.1 Performance Optimization

Backend Performance: - Async/await for I/O operations - Connection pooling for external APIs - Response caching strategies - Database query optimization

Frontend Performance: - Component-level caching - Lazy loading for large data - Optimized re-rendering - Progressive web app features

11.2 Scalability Design

Horizontal Scaling: - Stateless service design - Load balancer configuration - Database sharding strategies - Microservices decomposition

Vertical Scaling: - Resource monitoring and alerts - Auto-scaling triggers - Performance bottleneck identification - Capacity planning strategies

---

12. Monitoring and Observability

12.1 Logging Strategy

Structured Logging:

```
# Log entry structure
LOG_ENTRY = {
```

```
    "timestamp": "ISO8601",
    "level": "INFO|WARN|ERROR",
    "service": "service_name",
    "operation": "operation_name",
    "user_id": "session_id",
    "duration_ms": "execution_time",
    "status": "success|error",
    "metadata": {}
}
```

Log Categories: - **Application Logs**: Business logic and user interactions - **Performance Logs**: Response times and resource usage - **Security Logs**: Authentication and authorization events - **Error Logs**: Exceptions and system failures

12.2 Metrics and Monitoring

Key Performance Indicators: - Request throughput and latency - AI service response times - Error rates by service and endpoint - User engagement and satisfaction metrics

Monitoring Tools: - **Health Checks**: Built-in service health endpoints - **Application Metrics**: Custom metrics collection - **Infrastructure Monitoring**: Resource utilization tracking - **Alerting**: Automated incident response

---

13. Conclusion

SONA AI Assistant represents a comprehensive, production-ready AI platform designed with enterprise-grade architecture principles. The modular design enables rapid iteration and scaling while maintaining system reliability and performance.

Key Success Factors

1. **Modular Architecture**: Enables independent development and scaling of components
2. **AI Service Abstraction**: Allows for easy model switching and optimization
3. **Comprehensive Error Handling**: Ensures robust operation under various conditions
4. **Production Readiness**: Full containerization and cloud deployment support
5. **Extensible Design**: Supports future enhancements and feature additions

Expected Outcomes

- **User Experience**: Intuitive, responsive AI assistant with multi-modal interaction
- **Developer Experience**: Clean, maintainable codebase with comprehensive documentation
- **Operational Excellence**: Reliable, scalable system with comprehensive monitoring
- **Business Value**: Demonstrable AI capabilities with clear path to enhancement

---

Document Version: 1.0.0
Last Updated: 26 May 2025
Next Review: T.B.D