Helix CLI Specification v4.0 - Implementation-Ready Technical Blueprint

Executive Implementation Analysis

Based on comprehensive analysis of reference implementations (OpenCode, Charm Crush, Ollama Code, Codename Goose, Gemini Code, Qwen Code), this specification provides exact implementation details with proven architectural patterns.

Key Implementation Insights from Reference Projects:

- **OpenCode**: Clean Go architecture with panic recovery and structured logging
- **Codename Goose**: Rust-based with MCP (Model Context Protocol) integration and comprehensive tool ecosystem
- **Ollama Code**: TypeScript-based with sophisticated tool discovery and execution systems
- **Qwen Code**: Advanced provider abstraction with streaming and compression capabilities

Critical Success Factors Identified:

1. **Modular Tool Systems**: Reference projects demonstrate that successful AI coding assistants implement modular tool architectures with standardized interfaces
2. **Provider-Agnostic Interfaces**: All analyzed projects use abstraction layers to support multiple LLM providers seamlessly
3. **Robust Error Handling**: Comprehensive error recovery and graceful degradation are essential for production systems
4. **Streaming Support**: Real-time response streaming significantly improves user experience
5. **Hardware Optimization**: Automatic hardware detection and model optimization are critical for performance
6. **Comprehensive Testing**: 100% test coverage with multiple test types ensures reliability
7. **Security Architecture**: End-to-end encryption and access control are mandatory for enterprise use

Implementation Philosophy:

- **Terminal-First Design**: All features must work seamlessly in terminal environments
- **Progressive Enhancement**: Features scale from basic to advanced based on hardware capabilities
- **Zero Configuration Defaults**: Sensible defaults that work out of the box
- **Composable Architecture**: Independent components that can be used separately or together
- **Extensibility First**: Plugin system for unlimited customization and integration

Technical Foundation Requirements:

- **Language**: Go (Golang) for all core components
- **Architecture**: Microservices with bash orchestration
- **Data Storage**: PostgreSQL with SQLCipher encryption
- **Configuration**: JSON-based with environment variable overrides
- **Cross-Platform**: Native support for Linux, macOS, BSDs, Windows

Success Metrics:

- **Response Time**: Sub-second command execution (<500ms target)
- **Resource Efficiency**: Optimal hardware utilization (>85% target)
- **Test Coverage**: 100% automated test implementation
- **User Satisfaction**: High adoption and retention rates (>90% target)
- **Code Quality**: SonarQube A rating
- **Security**: Snyk vulnerability-free status
- **Reliability**: 99.9% uptime for core features
- **Documentation**: 100% coverage with multi-language support# 2. Core System Architecture - Implementation Details

2.1 Component Architecture Implementation

2.1.1 LLM Integration Layer - Exact Implementation

Interface Definition Requirements:

```
// Core LLM Provider Interface - Must implement exactly
type LLMProvider interface {
    // Generate must handle context cancellation and streaming
    Generate(ctx context.Context, req GenerationRequest)
(*GenerationResponse, error)

    // Stream must implement proper backpressure and chunk handling
    Stream(ctx context.Context, req GenerationRequest) (<-chan StreamChunk,
error)

    // GetCapabilities must return detailed provider-specific capabilities
    GetCapabilities() ProviderCapabilities

    // ValidateConfig must perform comprehensive configuration validation
    ValidateConfig() error

    // Close must ensure proper resource cleanup
    Close() error
}
```

Provider Implementation Patterns:

- **BaseProvider**: Common functionality for all providers
- **LlamaCPPProvider**: Local Llama.cpp integration with hardware optimization

- **OllamaProvider**: Ollama API client with model management
- **OpenRouterProvider**: OpenRouter API integration
- **DeepSeekProvider**: DeepSeek API client
- **QwenProvider**: Qwen API client
- **ClaudeProvider**: Anthropic Claude API
- **GeminiProvider**: Google Gemini API
- **GrokProvider**: xAI Grok API
- **MistralProvider**: Mistral AI API
- **HuggingFaceProvider**: HuggingFace endpoints
- **NvidiaProvider**: NVIDIA NIM API

2.1.2 User Interface Layer - Implementation Specifications

Terminal UI Component Structure:

```
type TerminalUI struct {
    app         *tview.Application // Must use tview for terminal UI
    layout      *LayoutManager    // Must implement fixed layout regions
    components  map[string]UIComponent
    eventBus    *EventBus         // Must use event-driven architecture
    state       *UIState          // Must maintain consistent UI state
}
```

Required UI Components:

- **HeaderComponent**: ASCII art + project info + system status
- **NavigationComponent**: Mode and project tree with keyboard navigation
- **InputComponent**: Command input with history and auto-completion
- **OutputComponent**: Results with syntax highlighting and pagination
- **StatusComponent**: System status and real-time notifications
- **HelpComponent**: Contextual help system with search

Layout Manager Specifications:

- **Header**: Top 3 lines - project name, mode, status indicators
- **Navigation**: Left 20% - project hierarchy and available modes
- **Main**: Center 60% - input/output with proper scrolling
- **Sidebar**: Right 20% - context and active tools
- **Status**: Bottom 2 lines - system metrics and notifications

2.1.3 Processing Engine Layer - Implementation Requirements

Base Engine Interface:

```
type ProcessingEngine interface {
    // Initialize must validate all dependencies and setup state
```

```
    Initialize(ctx *ProjectContext) error

    // Execute must handle cancellation and progress reporting
    Execute(req *ExecutionRequest) (*ExecutionResult, error)

    // Validate must perform comprehensive input validation
    Validate() []ValidationError

    // Cleanup must ensure proper resource release
    Cleanup() error

    // GetProgress must return detailed execution progress
    GetProgress() *ExecutionProgress
}
```

Engine Implementations:

- **PlannerEngine**: Project analysis and planning
- **BuilderEngine**: Code generation and modification
- **RefactorerEngine**: Code optimization and restructuring
- **TesterEngine**: Comprehensive testing execution
- **DebuggerEngine**: Issue detection and resolution
- **DesignerEngine**: Design system integration
- **DiagramEngine**: Diagram generation
- **DeploymentEngine**: Deployment automation
- **PortingEngine**: Cross-platform code conversion

2.2 Data Architecture - Implementation Specifications

2.2.1 Database Schema - Exact Implementation

Core Tables Structure:

```
-- Projects table must store exactly these fields
CREATE TABLE projects (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255) NOT NULL,
    path TEXT NOT NULL UNIQUE,
    config JSONB NOT NULL DEFAULT '{}',
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),

    -- Indexes must be created exactly as specified
    CONSTRAINT projects_name_check CHECK (name ~ '^[a-zA-Z0-9_-]+$'),
    CONSTRAINT projects_path_check CHECK (path ~ '^/[a-zA-Z0-9_/-]+$')
);

-- Sessions table must implement exactly this structure
CREATE TABLE sessions (
```

```
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    project_id UUID NOT NULL REFERENCES projects(id) ON DELETE CASCADE,
    name VARCHAR(255),
    state JSONB NOT NULL DEFAULT '{}',
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    active_until TIMESTAMPTZ,

    -- Session state management
    status VARCHAR(50) NOT NULL DEFAULT 'active' CHECK (status IN
('active', 'paused', 'completed', 'failed')),

    -- Indexes for performance
    CONSTRAINT sessions_name_check CHECK (name IS NULL OR name ~ '^[a-zA-
Z0-9_-]+$')
);

-- Additional required tables:
- requests: Command execution history and results
- models: Model registry with capabilities
- users: User management and preferences
- permissions: Access control and roles
- logs: System and audit logging
- configurations: Hierarchical configuration storage
```

2.2.2 Configuration Management - Implementation Requirements

Configuration Structure:

```
type Config struct {
    Global    GlobalConfig    `json:"global" validate:"required"`
    Providers ProviderConfigs `json:"providers" validate:"required,dive"`
    UI        UIConfig        `json:"ui" validate:"required"`
    Security  SecurityConfig  `json:"security" validate:"required"`
    Database  DatabaseConfig  `json:"database" validate:"required"`
}
```

Global Configuration Specifications:

```
type GlobalConfig struct {
    DefaultProvider string            `json:"default_provider"
validate:"required,oneof=llama-cpp ollama openrouter deepseek qwen claude
gemini grok mistral huggingface nvidia"`
    Theme           string            `json:"theme"
validate:"required,oneof=default warm-red blue yellow gold grey white
darcula dark-blue violet warm-orange"`
    AutoConfirm     bool              `json:"auto_confirm"`
    MaxWorkers      int               `json:"max_workers"
validate:"required,min=1,max=64"`
```

```
    LogLevel        string            `json:"log_level"
validate:"required,oneof=debug info warn error"`
    DataDir         string            `json:"data_dir"
validate:"required,dirpath"`
    CacheDir        string            `json:"cache_dir"
validate:"required,dirpath"`
}
```

Configuration Loading Process:

1. **Primary Source**: Load from `~/.config/Helix/helix.json`
2. **Environment Overrides**: Apply `HELIX_*` environment variables
3. **Validation**: Validate all configuration values against JSON schema
4. **Defaults**: Set sensible defaults for missing values
5. **Directory Creation**: Create required directories with proper permissions
6. **Backup**: Create backup of existing configuration before modification

Hierarchical Configuration Levels:

- **Global**: System-wide settings in `~/.config/Helix/helix.json`
- **Project**: Project-specific settings in `Helix.md` files
- **Module**: Module-specific configurations
- **Session**: Temporary session-specific settings
- **User**: User preferences and customization# 3. LLM Integration & Management - Implementation Details

3.1 Provider Implementation - Exact Requirements

3.1.1 Llama.cpp Integration - Implementation Specifications

Implementation Structure:

```
type LlamaCPPProvider struct {
    config     LlamaConfig
    process    *exec.Cmd
    apiClient  *http.Client
    modelPath  string
    context    *LlamaContext

    // Required implementation details:
    // - Must download Llama.cpp source from official repository
    // - Must compile with hardware-specific optimizations
    // - Must support model formats: GGUF, GGML, Safetensors
    // - Must implement automatic model conversion when needed
    // - Must handle context window management with sliding window
    // - Must implement memory-mapped loading for models >4GB
    // - Must support streaming with proper backpressure
    // - Must implement GPU memory management and fallback
}
```

Hardware-Specific Compilation Flags:

- **NVIDIA GPUs**: `-DGGML_USE_CUBLAS` for CUDA acceleration
- **Apple Silicon**: `-DGGML_USE_METAL` for Metal acceleration
- **AMD/Intel GPUs**: `-DGGML_USE_VULKAN` for Vulkan acceleration
- **CPU Optimization**: `-DGGML_USE_OPENBLAS` for CPU acceleration
- **Memory Management**: `-DGGML_USE_CPU_HBM` for high-bandwidth memory

Model Format Support:

- **GGUF**: Primary format with quantization support
- **GGML**: Legacy format with automatic conversion
- **Safetensors**: PyTorch format with conversion pipeline
- **ONNX**: Optional support for ONNX models
- **TensorFlow**: Optional support for TensorFlow models

3.1.2 Ollama Integration - Implementation Requirements

Implementation Structure:

```
type OllamaProvider struct {
    baseURL    string            // Must support both local and remote
instances
    httpClient *http.Client      // Must implement proper timeout and retry
logic
    models     []OllamaModel     // Must cache model list for performance

    // Required features:
    // - Must support local Ollama instance management
    // - Must implement model pull and management
    // - Must support custom model creation with Modelfile
    // - Must handle streaming generation with chunk processing
    // - Must implement system prompt templates
    // - Must support model quantization and optimization
    // - Must handle multiple concurrent requests
}
```

Model Management Features:

- **Automatic Discovery**: Detect available models and capabilities
- **Model Pull**: Download models with progress reporting
- **Integrity Verification**: Verify model integrity with checksums
- **Optimization**: Apply hardware-specific optimizations
- **Version Management**: Handle model updates and versioning
- **Custom Models**: Support for custom model creation

3.1.3 Remote API Providers - Implementation Specifications

API Configuration Structure:

```
type APIConfig struct {
    BaseURL    string            `json:"base_url" validate:"required,url"`
    APIKey     string            `json:"api_key" validate:"required"`
    Timeout    time.Duration     `json:"timeout"
validate:"required,min=1s,max=5m"`
    Headers    map[string]string `json:"headers"`
    RetryPolicy RetryPolicy      `json:"retry_policy" validate:"required"`

    // Rate limiting must be implemented exactly:
    RateLimit  RateLimitConfig   `json:"rate_limit" validate:"required"`

    // Cost tracking must be implemented:
    CostConfig CostConfig        `json:"cost_config" validate:"required"`
}
```

Common Features Across All API Providers:

- **Rate Limiting**: Token bucket algorithm with burst support
- **Request/Response Logging**: Structured logging with correlation IDs
- **Error Handling**: Exponential backoff with jitter
- **Cost Tracking**: Real-time cost monitoring and budget enforcement
- **Concurrent Requests**: Semaphore-based request management
- **Circuit Breaker**: Fault tolerance with automatic recovery
- **Request Deduplication**: Cache identical requests
- **Streaming Support**: Chunked response handling

3.2 Model Management System - Implementation Details

3.2.1 Hardware Detection & Analysis - Exact Implementation

Hardware Analyzer Structure:

```
type HardwareAnalyzer struct {
    GPUDetector    *GPUDetector
    MemoryAnalyzer *MemoryAnalyzer
    CPUAnalyzer    *CPUAnalyzer
    StorageChecker *StorageChecker
    NetworkChecker *NetworkChecker

    // Detection capabilities must include:
    // - GPU VRAM capacity and type (NVIDIA/AMD/Intel)
    // - System RAM availability and speed
    // - CPU cores, architecture, and capabilities
    // - Storage space, type (SSD/HDD), and speed
```

```
    // - Network bandwidth and latency
    // - Thermal and power constraints
    // - Operating system and kernel version
}
```

Hardware Detection Methods:

- **GPU Detection**: CUDA, ROCm, Metal API queries
- **Memory Analysis**: Available RAM and swap space
- **CPU Analysis**: Cores, architecture, instruction sets
- **Storage Analysis**: Free space, type, I/O performance
- **Network Analysis**: Bandwidth, latency, connectivity
- **Thermal Monitoring**: Temperature and power constraints

3.2.2 Model Selection Algorithm - Exact Implementation

Model Selection Process:

```
func (m *ModelManager) SelectOptimalModel(taskType TaskType, constraints
Constraints) (*Model, error) {
    // Step 1: Filter by hardware constraints
    availableModels := m.GetAvailableModels()
    filtered := m.FilterByHardware(availableModels, constraints)

    // Step 2: Score by task suitability
    scored := m.ScoreByTaskSuitability(filtered, taskType)

    // Step 3: Apply user preferences and history
    final := m.ApplyPreferences(scored)

    // Step 4: Return best match with fallback options
    if len(final) == 0 {
        return nil, fmt.Errorf("no suitable model found for task type %s",
taskType)
    }

    return final[0], nil
}
```

Task Suitability Scoring:

- **Planning Tasks**: Models with strong reasoning capabilities
- **Code Generation**: Models trained on code datasets
- **Testing Tasks**: Models with structured output capabilities
- **Debugging Tasks**: Models with analytical capabilities
- **Design Tasks**: Models with creative capabilities

3.2.3 Model Installation Process - Implementation Requirements

Installation Steps:

```
func (m *ModelManager) InstallModel(modelName string, provider string)
error {
    // Step 1: Source selection and validation
    source, err := m.SelectSource(modelName, provider)
    if err != nil {
        return fmt.Errorf("source selection failed: %w", err)
    }

    // Step 2: Format detection and conversion
    format, err := m.DetectFormat(source)
    if err != nil {
        return fmt.Errorf("format detection failed: %w", err)
    }

    // Step 3: Hardware-specific optimization
    optimized, err := m.OptimizeForHardware(format)
    if err != nil {
        return fmt.Errorf("optimization failed: %w", err)
    }

    // Step 4: Integrity verification
    if err := m.VerifyIntegrity(optimized); err != nil {
        return fmt.Errorf("integrity verification failed: %w", err)
    }

    // Step 5: Registration and metadata storage
    if err := m.RegisterModel(optimized); err != nil {
        return fmt.Errorf("registration failed: %w", err)
    }

    return nil
}
```

Installation Sources:

- **HuggingFace**: Primary source with authentication
- **OpenRouter**: Alternative source with API access
- **Direct Download**: From model provider URLs
- **Local Files**: From user-provided model files
- **Custom Sources**: User-defined download locations

3.3 Advanced LLM Features - Implementation Specifications

3.3.1 Tool Calling Implementation - Exact Requirements

Tool Call Structure:

```
type ToolCall struct {
    Name      string                 `json:"name" validate:"required"`
    Arguments map[string]interface{} `json:"arguments" validate:"required"`
    ID        string                 `json:"id" validate:"required,uuid"`

    // Tool execution must implement exactly:
    // - Parameter validation and type checking
    // - Permission checking and security validation
    // - Execution with timeout and cancellation
    // - Result collection and formatting
    // - Error handling and recovery
}
```

Available Tools Implementation:

- **FileSystemTool**: Read/write file operations with permission checks
- **GitTool**: Version control operations with conflict resolution
- **BuildTool**: Compilation and building with dependency management
- **TestTool**: Test execution with coverage reporting
- **DatabaseTool**: Database operations with transaction management
- **APITool**: HTTP API calls with authentication and retry
- **ShellTool**: Command execution with sandboxing
- **NetworkTool**: Network operations and connectivity testing

3.3.2 Thinking Process Support - Implementation Details

Thinking Process Structure:

```
type ThinkingProcess struct {
    Steps    []ThinkingStep `json:"steps" validate:"required,dive"`
    Context  *Context       `json:"context" validate:"required"`
    Decision *Decision      `json:"decision" validate:"required"`

    // Thinking step types must include exactly:
    // - Analysis: Problem breakdown and requirement gathering
    // - Research: Information gathering and knowledge synthesis
    // - Planning: Solution design and architecture planning
    // - Validation: Solution verification and testing planning
    // - Optimization: Performance improvement and refinement
}
```

Thinking Process Execution:

```
func (t *ThinkingEngine) ExecuteThinking(process *ThinkingProcess) error {
    // Step 1: Context setup and initialization
    if err := t.SetupContext(process.Context); err != nil {
```

```
        return fmt.Errorf("context setup failed: %w", err)
    }

    // Step 2: Execute thinking steps sequentially
    for i, step := range process.Steps {
        if err := t.ExecuteStep(step, i); err != nil {
            return fmt.Errorf("step %d execution failed: %w", i, err)
        }
    }

    // Step 3: Final decision and validation
    if err := t.FinalizeDecision(process.Decision); err != nil {
        return fmt.Errorf("decision finalization failed: %w", err)
    }

    return nil
}
```

Thinking Step Types:

- **Analysis Step**: Break down problems, identify requirements
- **Research Step**: Gather information, synthesize knowledge
- **Planning Step**: Design solutions, create architecture
- **Validation Step**: Verify solutions, plan testing
- **Optimization Step**: Improve performance, refine approach# 4. User Interface Implementation - Exact Specifications

4.1 Terminal UI Architecture - Implementation Requirements

4.1.1 UI Component Structure - Exact Implementation

Component Architecture:

```
type TerminalUI struct {
    app         *tview.Application // Must use tview for terminal UI
    layout      *LayoutManager    // Must implement fixed layout regions
    components  map[string]UIComponent
    eventBus    *EventBus         // Must use event-driven architecture
    state       *UIState          // Must maintain consistent UI state
}
```

Required Components Specifications:

HeaderComponent:

- **ASCII Art**: Dynamic project branding from Assets/Logo.png
- **Project Info**: Current project name, path, and status
- **System Status**: CPU, memory, and model usage indicators

- **Mode Indicator**: Current active mode (plan, build, test, etc.)

NavigationComponent:

- **Project Tree**: Hierarchical project structure navigation
- **Mode Selection**: Quick access to all available modes
- **Session Management**: Active sessions and quick switching
- **Keyboard Navigation**: Arrow keys and hotkey support

InputComponent:

- **Command Input**: Multi-line command input with syntax highlighting
- **History Management**: Command history with search and filtering
- **Auto-completion**: Context-aware command and parameter completion
- **Multi-modal Input**: Support for text, file, and clipboard input

OutputComponent:

- **Syntax Highlighting**: Language-specific code highlighting
- **Pagination**: Scrollable output with page navigation
- **Search Functionality**: Text search within output
- **Export Capabilities**: Save output to files or clipboard

StatusComponent:

- **Real-time Metrics**: CPU, memory, disk, network usage
- **Progress Indicators**: Current operation progress
- **Notifications**: System and user notifications
- **Error Reporting**: Error messages and warnings

HelpComponent:

- **Contextual Help**: Mode-specific help documentation
- **Command Reference**: Complete command documentation
- **Tutorial System**: Interactive tutorials and guides
- **Search Functionality**: Help content search

4.1.2 Layout Manager - Implementation Specifications

Layout Regions Definition:

```
type LayoutManager struct {
    root       *tview.Flex
    regions    map[string]*tview.Box
    focusChain []string
    themes     *ThemeManager
}
```

Fixed Layout Regions:

- **Header**: Top 3 lines
    - Line 1: ASCII art and project name
    - Line 2: Current mode and session info
    - Line 3: System status indicators
- **Navigation**: Left 20% of screen
    - Project hierarchy tree
    - Mode selection buttons
    - Session management panel
- **Main**: Center 60% of screen
    - Input area (top 30%)
    - Output area (bottom 70%)
    - Split view with adjustable divider
- **Sidebar**: Right 20% of screen
    - Context information
    - Active tools panel
    - Quick actions menu
- **Status**: Bottom 2 lines
    - Line 1: Progress bars and metrics
    - Line 2: Notifications and errors

Responsive Behavior:

- **Terminal Resize**: Automatic layout adjustment
- **Minimum Size**: Enforce minimum terminal dimensions
- **Focus Management**: Keyboard focus cycling between regions
- **Modal Dialogs**: Overlay dialogs for user interaction

4.2 Input System - Implementation Details

4.2.1 Command Processing Pipeline - Exact Implementation

Six-Stage Processing Pipeline:

Stage 1: Input Capture and Normalization

- Capture user input from multiple sources
- Normalize input format and encoding
- Handle special characters and escape sequences
- Validate input length and structure

Stage 2: Syntax Parsing and Validation

- Parse command syntax with proper tokenization

- Validate command structure and parameters
- Handle quoted strings and escape sequences
- Generate parse tree for execution planning

Stage 3: Context Resolution

- Resolve project and session context
- Apply configuration and permissions
- Load relevant project state and history
- Set up execution environment

Stage 4: Permission Checking

- Verify user permissions for requested operation
- Check file system access rights
- Validate API and network permissions
- Enforce security policies and constraints

Stage 5: Execution Planning

- Create execution plan with dependencies
- Allocate resources and set up monitoring
- Prepare rollback and recovery procedures
- Set up progress tracking and reporting

Stage 6: Result Handling

- Capture and format execution results
- Handle errors and exceptions gracefully
- Update project state and history
- Generate reports and notifications

4.2.2 Command Categories - Implementation Requirements

Project Commands:

```
/project create <name> [path]     # Create new project
/project switch <name|id>         # Switch active project
/project list [--active]          # List available projects
/project info [name]              # Show project information
/project config <get|set> <key> [value]  # Manage project configuration
/project import <path>            # Import existing project
/project export <format>          # Export project data
```

Session Commands:

```
/session start [name]             # Start new session
/session join <id|qr>             # Join existing session
```

```
/session list [--active]          # List active sessions
/session kill <id>                # Terminate session
/session pause <id>               # Pause session
/session resume <id>              # Resume paused session
/session detach <id>              # Detach from session
```

Mode Commands:

```
/mode plan [target]               # Enter planning mode
/mode build [module]              # Enter building mode
/mode test [--coverage]           # Enter testing mode
/mode refactor [scope]            # Enter refactoring mode
/mode debug [issue]               # Enter debugging mode
/mode design [component]          # Enter design mode
/mode diagram [type]              # Enter diagram mode
/mode deploy [profile]            # Enter deployment mode
/mode port [target]               # Enter porting mode
```

Model Commands:

```
/model list [--available|--installed]  # List models
/model install <name> [--provider]     # Install model
/model switch <name>                   # Switch active model
/model info <name>                     # Show model information
/model optimize <name>                 # Optimize model for hardware
/model remove <name>                   # Remove installed model
/model update <name>                   # Update model to latest version
```

Configuration Commands:

```
/config get <key>                  # Get configuration value
/config set <key> <value>          # Set configuration value
/config export [file]              # Export configuration
/config import <file>              # Import configuration
/config reset [scope]              # Reset configuration to defaults
/config validate                   # Validate current configuration
```

4.3 Theme System - Implementation Specifications

4.3.1 Theme Definition - Exact Implementation

Theme Structure:

```
type Theme struct {
    Name        string            `json:"name" validate:"required"`
    Colors      ColorScheme       `json:"colors" validate:"required"`
    Styles      StyleScheme       `json:"styles" validate:"required"`
    Fonts       FontScheme        `json:"fonts" validate:"required"`
    Layout      LayoutSettings    `json:"layout" validate:"required"`
}

type ColorScheme struct {
    Primary      string `json:"primary" validate:"required,hexcolor"`
    Secondary    string `json:"secondary" validate:"required,hexcolor"`
    Accent       string `json:"accent" validate:"required,hexcolor"`
    Background   string `json:"background" validate:"required,hexcolor"`
    Foreground   string `json:"foreground" validate:"required,hexcolor"`
    Success      string `json:"success" validate:"required,hexcolor"`
    Warning      string `json:"warning" validate:"required,hexcolor"`
    Error        string `json:"error" validate:"required,hexcolor"`
    Info         string `json:"info" validate:"required,hexcolor"`
}
```

Theme Features:

- **Hot Reloading**: Apply theme changes without restart
- **Custom Themes**: User-defined theme creation
- **Theme Export**: Export themes for sharing
- **Theme Validation**: Validate theme compatibility
- **Fallback Themes**: Automatic fallback for missing themes

4.3.2 Built-in Themes - Implementation Requirements

Default Theme Specifications:

- **Default**: Green-based with ASCII art header
  - Primary: #00FF00 (Bright Green)
  - Secondary: #008800 (Dark Green)
  - Background: #000000 (Black)
  - Foreground: #FFFFFF (White)

Additional Built-in Themes:

- **Warm Red**: Red/orange color scheme
- **Blue**: Cool blue tones
- **Yellow**: Bright and energetic
- **Gold**: Luxury and premium feel
- **Grey**: Minimal and professional
- **White**: Clean and modern
- **Darcula**: Dark theme inspired by IntelliJ
- **Dark Blue**: Deep blue background

- **Violet**: Purple-based theme
- **Warm Orange**: Orange and brown tones

Theme Consistency Requirements:

- **Color Mapping**: Consistent semantic color usage
- **Contrast Ratios**: WCAG AA compliance for accessibility
- **Terminal Compatibility**: Support for 256-color and true color terminals
- **Performance**: Efficient theme application and switching
- **Documentation**: Complete theme documentation and examples# 5. Development Workflows - Implementation Details

5.1 Planning Mode - Exact Implementation

5.1.1 Planning Process - Implementation Requirements

Seven-Step Planning Process:

Step 1: Project Analysis and Requirements Gathering

- Analyze existing codebase structure and dependencies
- Identify project requirements and constraints
- Gather user stories and use cases
- Document technical and business requirements

Step 2: Technology Stack Research and Selection

- Research available technologies and frameworks
- Evaluate technology compatibility and performance
- Select optimal technology stack
- Document technology selection rationale

Step 3: Architecture Design and Component Planning

- Design system architecture and component structure
- Define API contracts and data models
- Plan database schema and storage strategy
- Create component interaction diagrams

Step 4: Development Timeline Estimation

- Break down work into manageable tasks
- Estimate effort for each task
- Create development timeline
- Identify critical path and dependencies

Step 5: Resource Requirement Calculation

- Calculate hardware and software requirements
- Estimate team size and skill requirements
- Plan infrastructure and deployment needs

- Budget estimation and resource allocation

Step 6: Risk Assessment and Mitigation

- Identify potential risks and challenges
- Assess risk impact and probability
- Develop mitigation strategies
- Create contingency plans

Step 7: Documentation Generation

- Generate comprehensive project documentation
- Create API documentation and user guides
- Produce architecture and design documents
- Generate deployment and operations guides

5.1.2 Project Analysis - Implementation Specifications

Analysis Capabilities:

- **Codebase Structure**: Parse and analyze project structure
- **Dependency Analysis**: Identify internal and external dependencies
- **Technology Stack**: Detect programming languages and frameworks
- **Performance Characteristics**: Analyze code performance patterns
- **Security Vulnerabilities**: Identify security issues and risks
- **Code Quality Metrics**: Measure code complexity and quality
- **Documentation Coverage**: Assess documentation completeness
- **Test Coverage Analysis**: Evaluate test coverage and quality

Analysis Tools Integration:

- **Static Analysis**: Integrate with SonarQube, CodeClimate
- **Dependency Scanning**: Use Snyk, OWASP Dependency Check
- **Security Analysis**: Integrate security scanning tools
- **Performance Profiling**: Use profiling tools for performance analysis
- **Code Metrics**: Calculate cyclomatic complexity, maintainability index

5.2 Building Mode - Implementation Details

5.2.1 Parallel Building Architecture - Exact Implementation

Coordinator-Worker Architecture:

```
type ParallelBuilder struct {
    coordinator *BuildCoordinator
    workers     []*BuildWorker
    queue       *BuildQueue
    results     *ResultAggregator
```

```
    // Implementation requirements:
    // - Module-based work distribution
    // - Dependency-aware scheduling
    // - Resource-based worker limits
    // - Progress tracking and synchronization
    // - Conflict detection and resolution
}
```

Worker Management:

- **Worker Allocation**: Dynamic worker allocation based on system resources
- **Task Distribution**: Intelligent task distribution to available workers
- **Load Balancing**: Automatic load balancing across workers
- **Fault Tolerance**: Worker failure detection and recovery
- **Resource Monitoring**: Real-time resource usage monitoring

Conflict Resolution:

- **Dependency Detection**: Automatic dependency analysis
- **Conflict Identification**: Detect code and resource conflicts
- **Resolution Strategies**: Multiple conflict resolution strategies
- **Merge Coordination**: Coordinated code merging and integration
- **Rollback Capability**: Automatic rollback on conflicts

5.2.2 Code Generation Process - Implementation Requirements

Code Generation Workflow:

Phase 1: Requirements Analysis and Specification

- Parse user requirements and specifications
- Validate requirements completeness and consistency
- Generate detailed technical specifications
- Create acceptance criteria and test cases

Phase 2: Template Selection and Customization

- Select appropriate code templates
- Customize templates for specific requirements
- Apply project coding standards and conventions
- Generate template-specific configurations

Phase 3: Code Generation with Validation

- Generate code according to specifications
- Validate generated code syntax and structure
- Apply code formatting and style rules
- Generate unit tests and documentation

Phase 4: Formatting and Style Application

- Apply consistent code formatting
- Enforce coding standards and conventions
- Generate code comments and documentation
- Apply language-specific best practices

Phase 5: Integration Testing

- Generate integration tests
- Test component interactions
- Validate API contracts
- Performance and load testing

Phase 6: Documentation Generation

- Generate API documentation
- Create user guides and tutorials
- Produce deployment documentation
- Generate maintenance and operations guides

5.3 Testing Mode - Implementation Specifications

5.3.1 Comprehensive Testing Framework - Exact Implementation

Test Types and Execution:

Unit Tests:

- Isolated function and method testing
- Mock dependencies and external services
- Test edge cases and error conditions
- Measure code coverage and quality

Integration Tests:

- Test component interactions
- Validate API contracts and data flow
- Test database and external service integration
- Performance and scalability testing

End-to-End Tests:

- Complete system workflow testing
- User interface and interaction testing
- Cross-browser and cross-platform testing
- Real user scenario simulation

Automation Tests:

- UI automation and interaction testing

- API automation and validation
- Performance and load automation
- Security automation testing

Performance Tests:

- Load testing and stress testing
- Performance benchmarking
- Resource usage monitoring
- Scalability testing

Security Tests:

- Vulnerability scanning
- Penetration testing
- Security compliance testing
- Data protection testing

5.3.2 Quality Scanning Integration - Implementation Requirements

SonarQube Integration:

- **Code Quality Analysis**: Static code analysis and quality gates
- **Security Vulnerability Detection**: Identify security issues
- **Code Smell Detection**: Detect code smells and anti-patterns
- **Technical Debt Measurement**: Calculate and track technical debt
- **Quality Metrics**: Comprehensive code quality metrics

Snyk Integration:

- **Dependency Vulnerability Scanning**: Scan for vulnerable dependencies
- **License Compliance**: Check open source license compliance
- **Container Security**: Scan Docker images for vulnerabilities
- **Infrastructure Security**: Infrastructure as code security scanning
- **Continuous Monitoring**: Real-time security monitoring

Custom Scanners:

- **Performance Issues**: Custom performance analysis tools
- **Architecture Violations**: Architecture compliance checking
- **Code Style Violations**: Enforce coding standards
- **Dependency Vulnerabilities**: Custom dependency analysis
- **Security Compliance**: Industry-specific security standards

5.4 Refactoring Mode - Implementation Details

5.4.1 Refactoring Process - Exact Implementation

Refactoring Types:

Extract Method/Function:

- Identify code blocks for extraction
- Analyze dependencies and variable scope
- Create new method/function with proper parameters
- Update call sites with new method calls
- Generate comprehensive tests for extracted code

Inline Method/Function:

- Analyze method usage and dependencies
- Replace method calls with inline code
- Handle parameter passing and return values
- Update variable scope and visibility
- Remove obsolete method definitions

Move Method/Class:

- Analyze class relationships and dependencies
- Identify optimal class/method placement
- Update references and imports
- Handle access modifiers and visibility
- Update documentation and tests

Rename Variables/Functions:

- Semantic analysis for meaningful names
- Update all references consistently
- Handle scope and namespace considerations
- Update documentation and comments
- Validate naming conventions

Change Signature:

- Analyze method usage and dependencies
- Update parameter lists and return types
- Handle default values and optional parameters
- Update all call sites consistently
- Generate migration scripts if needed

Extract Interface:

- Identify common method signatures
- Create interface definitions
- Implement interface in relevant classes
- Update type declarations and dependencies
- Generate interface documentation

Pull Up/Push Down Members:

- Analyze inheritance hierarchy

- Move members to appropriate class levels
- Update access modifiers and visibility
- Handle overriding and implementation
- Update documentation and tests

Replace Conditional with Polymorphism:

- Identify conditional logic patterns
- Create polymorphic class hierarchy
- Replace conditionals with method calls
- Implement strategy or state patterns
- Generate comprehensive tests

5.4.2 Code Quality Improvements - Implementation Requirements

Quality Improvement Areas:

Performance Optimization:

- Algorithm optimization and complexity reduction
- Memory usage optimization and leak prevention
- Database query optimization
- Network and I/O performance improvements
- Caching strategy implementation

Memory Usage Reduction:

- Memory leak detection and prevention
- Efficient data structure selection
- Garbage collection optimization
- Memory pooling and reuse strategies
- Resource cleanup and disposal

Code Complexity Reduction:

- Cyclomatic complexity reduction
- Method and class size optimization
- Dependency reduction and simplification
- Conditional logic simplification
- Code duplication elimination

Dependency Cleanup:

- Unused dependency identification and removal
- Dependency version optimization
- Circular dependency resolution
- Dependency conflict resolution
- Build time optimization

Documentation Enhancement:

- API documentation generation and improvement
- Code comment quality enhancement
- User guide and tutorial creation
- Architecture documentation
- Maintenance and operations documentation

Security Improvements:

- Security vulnerability remediation
- Input validation and sanitization
- Authentication and authorization improvements
- Data protection and encryption
- Security compliance implementation

Maintainability Enhancements:

- Code organization and structure improvement
- Naming convention standardization
- Error handling and logging improvements
- Configuration management enhancement
- Deployment and operations improvements# 6. Session & Collaboration System - Implementation Details

6.1 Session Management - Exact Implementation

6.1.1 Session Lifecycle - Implementation Requirements

Session States and Transitions:

```
type SessionState string

const (
    SessionCreated    SessionState = "created"
    SessionActive     SessionState = "active"
    SessionPaused     SessionState = "paused"
    SessionResumed    SessionState = "resumed"
    SessionCompleted  SessionState = "completed"
    SessionFailed     SessionState = "failed"
    SessionTerminated SessionState = "terminated"
)
```

Session Lifecycle Management:

Creation:

- Generate unique session ID
- Initialize session state and context
- Set up session storage and persistence

- Configure session-specific settings
- Create session metadata and audit trail

Activation:

- Validate session prerequisites
- Allocate session resources
- Initialize session components
- Start session monitoring
- Update session state to active

Suspension:

- Capture current session state
- Preserve session context and memory
- Release temporary resources
- Update session state to paused
- Generate suspension checkpoint

Resumption:

- Validate session resumption conditions
- Restore session state from checkpoint
- Reallocate required resources
- Resume session monitoring
- Update session state to active

Termination:

- Complete pending operations
- Clean up session resources
- Generate session summary and reports
- Archive session data
- Update session state to completed/failed

6.1.2 Session Types - Implementation Specifications

Single-User Sessions:

- **Isolated Execution**: Complete isolation from other sessions
- **Resource Management**: Dedicated resource allocation
- **State Persistence**: Automatic state saving and restoration
- **Error Isolation**: Failures contained within session
- **Performance Optimization**: Optimized for single-user performance

Multi-User Collaborative Sessions:

- **Real-time Synchronization**: WebSocket-based state synchronization
- **Conflict Resolution**: Automatic conflict detection and resolution
- **Access Control**: Granular permission management

- **Presence Awareness**: Real-time user presence and activity
- **Collaboration Tools**: Shared editing and communication features

Automated Background Sessions:

- **Scheduled Execution**: Time-based or event-triggered execution
- **Resource Optimization**: Low-priority resource allocation
- **Progress Monitoring**: Background progress tracking
- **Error Handling**: Automatic error recovery and notification
- **Result Collection**: Automated result aggregation and reporting

Detached Sessions:

- **Background Execution**: Continue execution without UI interaction
- **State Persistence**: Automatic state checkpointing
- **Resource Management**: Dynamic resource allocation
- **Progress Reporting**: Asynchronous progress updates
- **Reattachment Support**: Seamless reconnection to detached sessions

6.2 Multi-User Collaboration - Implementation Details

6.2.1 Real-time Synchronization - Exact Implementation

WebSocket-Based Architecture:

```
type WebSocketManager struct {
    connections map[string]*WebSocketConnection
    broadcast   chan BroadcastMessage
    sync        *StateSynchronizer
    conflicts   *ConflictResolver

    // Implementation requirements:
    // - Real-time state synchronization
    // - Conflict detection and resolution
    // - Change propagation and consistency
    // - Presence awareness and user tracking
    // - Permission management and access control
}
```

State Synchronization:

- **Delta Updates**: Only transmit changed state portions
- **Conflict Detection**: Automatic conflict identification
- **Merge Strategies**: Multiple conflict resolution strategies
- **Consistency Guarantees**: Eventual consistency with conflict resolution
- **Performance Optimization**: Efficient state compression and transmission

Change Propagation:

- **Event Broadcasting**: Real-time event distribution
- **Order Preservation**: Maintain operation order consistency
- **Causality Tracking**: Track operation dependencies
- **Rollback Support**: Automatic rollback on conflicts
- **Audit Trail**: Complete operation history and tracking

6.2.2 Access Control - Implementation Requirements

Role-Based Permissions:

```
type Permission struct {
    Resource    string   `json:"resource" validate:"required"`
    Action      string   `json:"action" validate:"required,oneof=read write
execute delete admin"`
    Conditions []string `json:"conditions"`
    Effect      string   `json:"effect" validate:"required,oneof=allow
deny"`
}

type Role struct {
    Name        string       `json:"name" validate:"required"`
    Permissions []Permission `json:"permissions" validate:"required,dive"`
    Inherits    []string     `json:"inherits"`
}
```

Permission Levels:

- **Project-Level Access**: Control access to entire projects
- **Session-Level Restrictions**: Limit session participation
- **Operation-Level Authorization**: Control specific operations
- **Resource-Level Permissions**: Granular resource access control
- **Temporal Access**: Time-limited access grants

Access Control Features:

- **Permission Inheritance**: Role hierarchy and permission inheritance
- **Temporary Grants**: Time-limited access permissions
- **Audit Logging**: Complete access and operation logging
- **Permission Templates**: Reusable permission templates
- **Bulk Management**: Efficient permission management at scale

6.3 Distributed Computing - Implementation Specifications

6.3.1 Node Management - Exact Implementation

Worker Node Architecture:

```
type WorkerNode struct {
    ID           string             `json:"id" validate:"required"`
    Capabilities NodeCapabilities `json:"capabilities" validate:"required"`
    Status       NodeStatus         `json:"status" validate:"required"`
    Resources    NodeResources      `json:"resources" validate:"required"`
    Load         NodeLoad           `json:"load" validate:"required"`
}
```

Node Registration and Discovery:

- **Automatic Discovery**: Automatic node detection and registration
- **Capability Reporting**: Detailed capability and resource reporting
- **Health Monitoring**: Continuous health and status monitoring
- **Load Balancing**: Intelligent load distribution across nodes
- **Fault Detection**: Automatic failure detection and recovery

Resource Management:

- **Dynamic Allocation**: Dynamic resource allocation based on demand
- **Resource Optimization**: Optimal resource utilization
- **Performance Monitoring**: Real-time performance tracking
- **Capacity Planning**: Predictive capacity planning
- **Cost Optimization**: Cost-effective resource allocation

6.3.2 Work Distribution - Implementation Requirements

Task Partitioning:

- **Dependency Analysis**: Automatic dependency detection
- **Task Granularity**: Optimal task size determination
- **Resource Matching**: Task-to-resource capability matching
- **Load Balancing**: Even distribution of computational load
- **Priority Management**: Task priority and scheduling

Result Aggregation:

- **Result Collection**: Efficient result gathering from workers
- **Progress Tracking**: Real-time progress monitoring
- **Error Handling**: Comprehensive error handling and recovery
- **Result Validation**: Automatic result validation and verification
- **Performance Analysis**: Performance metrics and optimization

Synchronization Mechanisms:

- **Barrier Synchronization**: Coordinate task completion
- **Checkpoint Coordination**: Distributed checkpoint management
- **State Synchronization**: Consistent state across nodes

- **Progress Synchronization**: Unified progress tracking
- **Error Synchronization**: Coordinated error handling

Resource Optimization:

- **Load Prediction**: Predictive load forecasting
- **Resource Allocation**: Dynamic resource allocation
- **Cost Management**: Cost-effective resource utilization
- **Performance Optimization**: Continuous performance improvement
- **Scalability Management**: Automatic scaling based on demand

Cost Management:

- **Cost Tracking**: Real-time cost monitoring
- **Budget Enforcement**: Budget limits and enforcement
- **Cost Optimization**: Cost-effective resource allocation
- **Usage Reporting**: Detailed usage and cost reporting
- **Billing Integration**: Integration with billing systems