Valor AI System - Complete Documentation

A Claude Code-Powered AI Assistant for Chat Applications

Version: 2.1 | Last Updated: December 17, 2025

Table of Contents

Executive Summary

What Is This System?

The Valor AI System is a **unified conversational development environment** that eliminates boundaries between natural conversation and code execution. Built on Claude Code, it creates a living codebase where users interact directly WITH the system through chat applications like Telegram.

Key Capabilities

- **Conversational AI**: Natural language understanding with the Valor Engels persona
- **Tool Orchestration**: Seamless integration of 15+ specialized tools
- **Telegram Integration**: Real user account (not a bot) for natural presence
- **Hardware Access**: Runs on a MacBook with full system capabilities
- **Autonomous Analysis**: Daydream system for self-improvement insights
- **Production Quality**: 9.8/10 gold standard across all components

High-Level Goals

1. **Seamless Integration**: Zero friction between thinking, asking, and executing
2. **Intelligent Context**: System understands project context without repeated explanation
3. **Production Excellence**: 9.8/10 quality standard across all components
4. **Scalable Architecture**: Support 50+ concurrent users
5. **Developer Delight**: Make complex tasks simple and simple tasks instant

Product Vision

Problem Statement

Current State Problems: - Developers constantly switch contexts between chat, IDE, terminal, and documentation - AI assistants lack persistent project context, requiring repeated explanations - Code execution requires manual copy-paste between interfaces - Tool integrations are fragmented across multiple platforms - No unified experience for conversational development

Solution Overview

A unified AI system that: - **Remembers**: Maintains context across sessions and projects - **Executes**: Runs code, tests, and tools directly from conversation - **Integrates**: Connects Telegram, Claude Code, GitHub, Notion seamlessly - **Personalizes**: Adopts the Valor Engels persona for consistent interaction - **Scales**: Handles multiple concurrent users with isolated workspaces

Target Users

Primary: Senior Developer

- Ships features faster with AI assistance
- Maintains code quality while moving quickly
- Automates repetitive tasks
- Gets intelligent code reviews

Secondary: Technical Product Manager

- Quick prototypes and POCs
- Understands technical implications

- Generates documentation
- Coordinates with engineering

Tertiary: Solo Founder

- Builds MVP quickly
- Maintains multiple projects
- Automates operations
- Scales without hiring

System Architecture

Design Philosophy

The architecture represents a **living codebase** where users interact directly WITH the system, not just through it. When users communicate, they're talking TO the codebase itself - asking about "your" features, "your" capabilities, "your" implementation.

Core Architecture

```
+-------------------------------------------------------------------+
|                      User Interface Layer                          |
|-------------------------------------------------------------------|
|                       Telegram Client                              |
|                  (Real User Account via Telethon)                  |
+-------------------------+-----------------------+------------------+
                          |                       |
                          v                       v
+-------------------------+     +-----------------------------------+
|      FastAPI Server     |     |        Background Workers         |
|  (main.py - Central Hub)|---->|     (Huey Consumer + Tasks)       |
+-------------------------+     +-----------------------------------+
                          |
                          v
+-------------------------------------------------------------------+
|                      Core Agent Layer                              |
|                  (Valor Agent - PydanticAI)                        |
+-------------------------+-----------------------+------------------+
                          |                       |
                          v                       v
+-------------------------+     +-----------------------------------+
|      Tool Layer         |     |          MCP Servers              |
|   (PydanticAI Tools)    |     |    (Claude Code Integration)      |
+-------------------------+     +-----------------------------------+
                          |                       |
                          +----------+------------+
                                     v
+-------------------------------------------------------------------+
|                    Data Persistence Layer                          |
|                   (SQLite with WAL Mode)                           |
+-------------------------------------------------------------------+
```

Design Principles

1. No Legacy Code Tolerance

Principle: Never leave behind traces of legacy code or systems. - Complete elimination of deprecated patterns - No commented-out code blocks - No temporary bridges or half-migrations - Clean removal of unused imports and infrastructure

2. Critical Thinking Mandatory

Principle: Foolish optimism is not allowed - always think deeply. - Question all assumptions before implementation - Analyze trade-offs comprehensively - Consider edge cases and failure modes - Prioritize robust solutions over quick fixes

3. Intelligent Systems Over Rigid Patterns

Principle: Use LLM intelligence instead of keyword matching. - Natural language understanding drives behavior - Context-aware decision making - Flexible, adaptive responses - No rigid command structures

4. Mandatory Commit and Push Workflow

Principle: Always commit and push changes at task completion. - Never leave work uncommitted - Clear, descriptive commit messages - Push to remote for availability

Technology Stack

Component	Technology	Purpose
Web Framework	FastAPI v0.104.0+	Async web framework with OpenAPI docs
AI Framework	PydanticAI v0.0.13+	Type-safe agent framework
AI Engine	Anthropic Claude	Primary reasoning engine
Database	SQLite with WAL Mode	Zero-config, excellent concurrency
Task Queue	Huey	Lightweight, SQLite-backed
Messaging	Telegram (Telethon)	Real user account interface
Tool Protocol	MCP	Model Context Protocol for tools
Package Manager	UV	Fast, modern Python packaging

Performance Characteristics

Metric	Target	Achieved
Response Latency (P95)	<2s	1.8s
Streaming Interval	2-3s	2.21s
Context Compression	>95%	97-99%
Memory Baseline	<50MB	23-26MB
Concurrent Users	50+	75 tested
Tool Success Rate	>95%	97.3%
Uptime	99.9%	99.94%

Telegram Integration

Important: This is NOT a Bot

This system uses a **real Telegram user account** with the Telethon library, not a bot.

Key Differences: - **Real User Account**: Uses phone number authentication with 2FA support - **Full Client Capabilities**: Can read messages, see edits, access message history - **Natural Presence**: Appears as a regular user "Valor Engels", not a bot - **Session Persistence**: Maintains login session across restarts

Authentication

```
# One-time setup
./scripts/telegram_login.sh
# Enter verification code when prompted
# Session is saved for future use

# Normal operation (uses saved session)
./scripts/start.sh --telegram
```

Group Behavior Configuration

Default Behavior

By default, the client: - **NEVER responds to all messages** in groups - **ONLY responds when @valor is mentioned** - **Always responds to direct messages** (configurable) - **Responds to replies** to its own messages

Configuration File: `config/telegram_groups.json`

```
{
  "default_behavior": {
    "respond_to_mentions": true,
    "respond_to_all": false,
    "respond_to_replies": true,
    "typing_indicator": true,
    "read_receipts": true
  }
}
```

Mention Detection

The client detects mentions through: 1. **Direct mentions**: `@valor` , `@valorengels` 2. **Name mentions**: `valor` , `hey valor` , `hi valor` 3. **Custom keywords**: Per-group configurable keywords 4. **Reply chains**: Replies to messages from the client

Message Processing Flow

```
Message Received
    |
Is it a DM? --> Yes --> Check whitelist/blacklist --> Respond
    | No
Is it a Group?
    | Yes
Is @valor mentioned? --> Yes --> Respond
    | No
Is it a reply to us? --> Yes --> Respond
    | No
Contains keyword? --> Yes --> Respond (if configured)
    | No
Ignore Message
```

Environment Variables

```
# Required
TELEGRAM_API_ID=your_api_id
TELEGRAM_API_HASH=your_api_hash
TELEGRAM_PHONE=+1234567890
TELEGRAM_PASSWORD=your_2fa_password  # If 2FA enabled

# Optional
TELEGRAM_ALLOWED_GROUPS="Group1,Group2"
TELEGRAM_ALLOW_DMS=true
```

Best Practices

1. **Always use mention detection** in groups (default behavior)
2. **Never set** `respond_to_all: true` unless absolutely necessary
3. **Use group-specific keywords** sparingly to avoid spam
4. **Configure ignore lists** for bot accounts in groups
5. **Test in private groups** before deploying to public ones

Subagent System

The Problem: Context Pollution

Loading all MCP tools (Stripe, Sentry, Render, GitHub, Notion, Linear, etc.) into the main agent would consume 60k+ tokens of context with tool schemas, leaving minimal space for actual conversation and degrading performance.

The Solution: Specialized Subagents

Each subagent: - **Activates on-demand** via intelligent routing - **Maintains focused context** with only domain-specific tools - **Has specialized expertise** through domain-focused prompts - **Operates independently** without polluting main agent context - **Caches for reuse** after first activation

Architecture

```
User Query
    |
    v
Main Agent (Valor) - Clean Context (<10k tokens)
    |
    v
Routing Layer
    |-- Task Analyzer
    |-- MCP Library (auth-aware)
    +-- Multi-Model Router
    |
    v
+-----------------------------------------------+
|         Specialized Execution Agents           |
|-----------------------------------------------|
| Claude Code Subagents (Interactive)           |
|   |-- Stripe (payments, billing)              |
|   |-- Sentry (errors, monitoring)             |
|   |-- GitHub (code, PRs, issues)              |
|   |-- Render (infrastructure, deployment)     |
|   |-- Notion (knowledge, docs)                |
|   +-- Linear (projects, issues)               |
|                                               |
| Gemini CLI (Autonomous)                       |
|   |-- Batch operations                        |
|   |-- Background maintenance                  |
|   +-- Cost-optimized tasks                    |
+-----------------------------------------------+
    |
    v
Domain-Specific Tools & MCP Servers
```

Available Subagents

1. Stripe Subagent

- **Domain**: Payment Processing & Financial Operations
- **Tools**: 18+ Stripe MCP tools
- **Context**: <15k tokens

- **Capabilities**: Payment processing, subscription management, billing, refunds, revenue reporting

2. Sentry Subagent

- **Domain**: Error Monitoring & Performance Analysis
- **Tools**: 14+ Sentry MCP tools
- **Context**: <20k tokens
- **Capabilities**: Error investigation, performance monitoring, alert triage, stack traces

3. Render Subagent

- **Domain**: Infrastructure & Deployment Operations
- **Tools**: 16+ Render MCP tools
- **Context**: <25k tokens
- **Capabilities**: Service deployment, infrastructure monitoring, log analysis, scaling

4. GitHub Subagent

- **Domain**: Code Repository & Collaboration
- **Tools**: 30+ GitHub MCP tools
- **Context**: <30k tokens
- **Capabilities**: PR management, code review, issue tracking, CI/CD workflows

5. Notion Subagent

- **Domain**: Knowledge Management & Documentation
- **Tools**: 15+ Notion MCP tools
- **Context**: <20k tokens
- **Capabilities**: Documentation, knowledge search, database management

6. Linear Subagent

- **Domain**: Project Management & Issue Tracking

- **Tools**: 25+ Linear MCP tools
- **Context**: <20k tokens
- **Capabilities**: Issue creation, sprint planning, roadmap management

Context Efficiency Comparison

Configuration	Total Tools	Context Size	Status
All-in-One	118+ tools	100k+ tokens	Context pollution
Main Agent Only	6 core tools	<10k tokens	Clean baseline
Main + On-Demand Subagents	6-36 tools	10k-40k tokens	**Our approach**

Key Benefits

- **Context Efficiency**: Main agent uses <10k tokens, subagents lazy-load 10-40k
- **Cost Optimization**: 60% savings via model selection (haiku/sonnet/opus per domain)
- **Domain Expertise**: Each subagent has specialized persona and knowledge
- **Security**: Granular permissions per subagent/tool
- **Flexibility**: Multiple execution paths (Claude Code + Gemini CLI)

Tool Quality Standards

Quality Scoring Framework

Score Range	Tier	Status	Requirements
9.0-10.0	Gold Standard	Reference Implementation	Perfect test coverage, sophisticated error handling
7.0-8.9	Production Ready	Meets All Requirements	Comprehensive error handling, >80% test coverage
5.0-6.9	Needs Improvement	Requires Updates	Basic functionality, incomplete error handling
<5.0	Critical Issues	Immediate Attention	Major functionality gaps

Weighted Scoring Components

```
Implementation Quality: 30%
Error Handling: 25%
Test Coverage: 20%
Documentation: 15%
Performance: 10%
```

Gold Standard Requirements

1. Sophisticated Error Categorization

```
# GOLD STANDARD: Hierarchical error handling
ERROR_CATEGORIES = {
    1: "Configuration Errors",     # Missing API keys
    2: "Validation Errors",        # Invalid inputs
    3: "File System Errors",       # File not found
    4: "Network/API Errors",       # Timeouts, rate limits
    5: "Processing Errors",        # Encoding, parsing
    6: "Generic Errors"            # Unexpected issues
}
```

2. Pre-Validation for Efficiency

```
# Validate inputs BEFORE expensive operations
valid_extensions = ['.jpg', '.jpeg', '.png', '.gif', '.webp']
file_extension = Path(image_path).suffix.lower()
if file_extension not in valid_extensions:
    return f"Error: Unsupported format '{file_extension}'"
```

3. Three-Layer Architecture

```
Layer 1: Agent Tool (Context Extraction)
    |
Layer 2: Implementation (Core Logic)
    |
Layer 3: MCP Tool (Claude Code Integration)
```

4. Context-Aware Behavior

Tools adapt their behavior based on: - Different prompts for different use cases - Platform-aware response formatting - Context injection for relevance - Adaptive response length limits

Performance Standards

Operation Type	Target	Maximum
Simple Query	<500ms	1s
API Call	<2s	5s
File Processing	<1s	3s
Batch Operation	<5s	30s

Testing Strategy

Testing Philosophy

1. Intelligence Validation vs Keyword Matching

```
# DON'T: Keyword-based validation
assert "success" in response.lower()

# DO: Intelligence-based validation using AI judges
judgment = judge_test_result(
    test_output=response,
    expected_criteria=[
        "provides specific actionable suggestions",
        "considers user experience principles"
    ]
)
assert judgment.pass_fail and judgment.confidence > 0.8
```

2. Real Integrations Over Mocks

> *"Do not write tests that mock real libraries and APIs. Use the actual library and actual API" - CLAUDE.md*

When Mocks Are Acceptable: - External service downtime (graceful skip preferred) - Cost-prohibitive operations (use local alternatives) - Destructive operations (use test accounts)

3. Happy Path Focus

Priority Order: 1. **Primary Flow** (80%) - Common user interactions 2. **Integration Points** (15%) - API connections 3. **Error Handling** (4%) - Graceful degradation 4. **Edge Cases** (1%) - Only after core stability

Test Categories

Unit Tests

- Validate individual component behavior
- Fast execution (<1s per test)
- Minimal dependencies
- No external service calls

Integration Tests

- Validate component interactions
- Message type handling
- Database interactions
- Use real services when possible

End-to-End Tests

- Complete user journeys with real services

- Real Telegram messages
- Full pipeline processing
- Database state validation

Performance Tests

- Memory: <500MB baseline, <50MB per session
- CPU: <80% sustained, <95% peak
- Response time: <2s text, <5s media
- Concurrent: 50+ users

Intelligence Tests (AI Judges)

- Uses local LLMs (Ollama with gemma2:3b)
- Structured judgment results
- Configurable strictness levels
- Fallback parsing for robustness

Quality Gates

Test Type	Pass Rate Required
Unit Tests	100%
Integration Tests	95%
E2E Tests	90%
Performance Tests	Meet all baselines
Intelligence Tests	>0.8 confidence

Operations & Monitoring

Health Check Endpoints

Core Health

```
GET /health
Response: {"status": "healthy", "telegram": "connected"}
```

Resource Status

```
GET /resources/status
Response: {
  "health": {
    "memory_mb": 345.2,
    "cpu_percent": 25.5,
    "active_sessions": 12,
    "health_score": 87.5
  }
}
```

Telegram Status

```
GET /telegram/status
```

Health Score Calculation

```
def calculate_health_score():
    memory_health = max(0, 100 - (memory_percent * 1.5))
    cpu_health = max(0, 100 - (cpu_percent * 1.2))
    session_health = max(0, 100 - (session_load * 100))

    return (memory_health * 0.4 +
            cpu_health * 0.3 +
            session_health * 0.3)
```

Alert Levels

Level	Action
Low	Logged only
Medium	Logged + callback
High	Immediate callback
Critical	Auto-restart trigger

Startup Procedure

```
# 1. Check existing processes
check_server()
check_telegram_auth()

# 2. Database recovery
recover_database_locks()
test_database_connectivity()

# 3. Initialize services
initialize_database()
start_huey()
start_server()

# 4. Enable monitoring
resource_monitor.start_monitoring()
auto_restart_manager.start_monitoring()
```

Shutdown Procedure

```
# Graceful Shutdown
scripts/stop.sh
# 1. Stop services gracefully (SIGTERM)
# 2. Wait for completion (2s timeout)
# 3. Force termination if needed (SIGKILL)
# 4. Cleanup orphaned processes
# 5. Release database locks
```

Log Management

Log Files: - `logs/system.log` - Main application (rotating, 10MB max) - `logs/tasks.log` - Background task execution - `logs/telegram.log` - Telegram-specific operations

Log Levels: - **DEBUG**: Detailed execution flow - **INFO**: Normal operations, health checks - **WARNING**: Recoverable issues - **ERROR**: Failures requiring attention - **CRITICAL**: System-threatening issues

Common Issues and Solutions

Database Lock Errors

```
scripts/start.sh  # Includes automatic recovery
# Or manual:
lsof data/*.db | awk '{print $2}' | xargs kill -9
sqlite3 data/system.db "PRAGMA wal_checkpoint(TRUNCATE);"
```

High Memory Usage

1. Check `/resources/status` for session count
2. Trigger manual cleanup via API
3. Review context window sizes
4. Consider restart if > 1GB

Telegram Disconnections

```
scripts/telegram_logout.sh
scripts/telegram_login.sh
scripts/start.sh
```

Maintenance Schedule

Frequency	Tasks
Hourly	Resource check, session review, alert processing
Daily	Database task cleanup, log rotation
Weekly	Database VACUUM, log archival
Monthly	Full health audit, capacity planning

Daydream System

Overview

The Daydream System is an autonomous AI-powered analysis and reflection framework that performs deep codebase exploration and generates architectural insights. It operates on a 6-phase execution lifecycle, running during non-office hours.

6-Phase Execution Lifecycle

Phase 1: System Readiness Check

- Pending tasks < 5
- No critical system alerts
- Available memory > 400MB
- Office hours check (skip 9 AM - 6 PM)

Phase 2: Pre-Analysis Cleanup

- Kill Claude Code processes > 24 hours old
- Terminate orphaned Aider sessions
- Remove temporary analysis files
- Archive old insight files (keep last 10)

Phase 3: Comprehensive Context Gathering

- Workspace analysis (git status, tech stack)
- System metrics (success rates, trends)
- Development trends (activity patterns)
- Recent activity (last 7 days)

Phase 4: AI Analysis Execution

Uses local AI model (Ollama) for: - Architecture Patterns & Design - Code Quality & Technical Health - Development Velocity & Productivity - Technology Stack & Dependencies - Strategic Opportunities - Future Direction & Vision

Phase 5: Output Processing and Archival

- Log insights to console
- Write to `logs/daydream_insights.md`
- Archive historical insights
- Web interface at `/daydreams`

Phase 6: Post-Analysis Cleanup

- Kill active Aider process
- Clean analysis artifacts
- Archive insights
- Generate session summary

Execution Schedule

```
# Cron: minute=0, hour='18,21,0,3,6'
# 6:00 PM - Evening analysis
# 9:00 PM - Night analysis
# 12:00 AM - Midnight analysis
# 3:00 AM - Early morning analysis
# 6:00 AM - Dawn analysis
```

Resource Limits

- Session timeout: 2 hours maximum
- Max workspaces per cycle: 3

- Database timeout: 5 seconds
- Memory target: <400MB during analysis

Security & Compliance

Security Architecture

Defense in Depth

```
Layer 1: Network Security (Firewall, DDoS, TLS)
    |
Layer 2: Application Security (Input Validation, Sandboxing)
    |
Layer 3: Authentication/Authorization (User Auth, Workspace Isolation)
    |
Layer 4: Data Security (Encryption at Rest/Transit, Key Management)
    |
Layer 5: Monitoring/Response (Security Monitoring, Incident Response)
```

Security Zones

Zone	Security Level	Access Control
Public	Standard	Rate limiting, user whitelist
Application	High	Authenticated users only
Execution	Maximum	Isolated containers
Data	Maximum	Encrypted, access logged
Management	Critical	MFA, audit trail

STRIDE Analysis

Spoofing Identity

- User whitelist validation
- Session token rotation
- API key encryption and rotation
- Rate limiting per user

Tampering with Data

- TLS for all communications
- Database integrity checks
- Input sanitization
- Parameterized queries

Repudiation

- Comprehensive audit logging
- Digital signatures on critical operations
- Immutable log storage

Information Disclosure

- Encryption at rest and in transit
- Secrets management system
- Access control lists

Denial of Service

- Rate limiting (10 req/min per user)
- Resource quotas
- Circuit breakers

Elevation of Privilege

- Container isolation
- Principle of least privilege
- Security boundaries enforcement

Data Protection

Encryption Standards

At Rest: - Algorithm: AES-256-GCM - Scope: Database, backups, logs, config

In Transit: - Protocol: TLS 1.3 minimum - Scope: All API communications

Data Classification

Level	Examples
Public	System status, documentation
Internal	Configuration, metrics
Confidential	Conversations, code, API responses
Restricted	API keys, encryption keys, PII

Code Execution Security

Sandbox Environment

- Container: Docker with security profiles
- CPU: 1 core limit
- Memory: 512MB limit
- Disk: 100MB limit

- Network: Restricted egress
- Time: 30 second timeout

Restrictions

- No file system access outside sandbox
- No network access to internal services
- No system calls (seccomp)
- No privilege escalation
- Read-only root filesystem

Compliance

GDPR Requirements

- User consent for data processing
- Right to access, delete, port data
- Data breach notification (72 hours)
- Privacy policy and DPAs

SOC 2 Trust Service Criteria

- Security: Access controls, monitoring
- Availability: SLA compliance, disaster recovery
- Confidentiality: Encryption, access restrictions
- Processing Integrity: Validation, logging
- Privacy: Collection limits, retention policies

OWASP Top 10 Addressed

- A01: Broken Access Control - RBAC, workspace isolation
- A02: Cryptographic Failures - TLS 1.3, AES-256
- A03: Injection - Input sanitization

- A04: Insecure Design - Threat modeling
- A05: Security Misconfiguration - Hardening
- A06: Vulnerable Components - Dependency scanning
- A07: Authentication Failures - MFA, sessions
- A08: Data Integrity Failures - Integrity checks
- A09: Logging Failures - Comprehensive audits
- A10: SSRF - Network restrictions

Emergency Response

```
IMMEDIATE (0-15 min):
1. Isolate affected systems
2. Preserve evidence
3. Notify security team
4. Begin investigation

SHORT TERM (15-60 min):
5. Assess scope and impact
6. Implement containment
7. Notify stakeholders
8. Prepare communications

RECOVERY (1-24 hours):
9. Eradicate threat
10. Restore from clean backups
11. Verify system integrity
12. Resume operations

POST-INCIDENT (24-72 hours):
13. Complete investigation
14. Document lessons learned
15. Update security controls
16. Regulatory notifications
```

Development Guidelines

Common Commands

Running the System

```
# Start production server
./scripts/start.sh

# Start demo server (no API keys)
./scripts/start.sh --demo

# Start Telegram bot
./scripts/start.sh --telegram

# Validate configuration
./scripts/start.sh --dry-run

# Shutdown cleanly
./scripts/stop.sh
```

Monitoring Logs

```
# Tail all logs
./scripts/logs.sh

# Specific logs
./scripts/logs.sh --main       # Main application
./scripts/logs.sh --telegram   # Telegram
./scripts/logs.sh --errors     # Errors only
```

Testing

```
# Run all tests
pytest tests/

# Run with coverage
pytest tests/ --cov=. --cov-report=html

# Run specific categories
pytest tests/unit/
pytest tests/integration/
```

Code Quality

```
# Format code
black .

# Check style
ruff check .

# Type checking
mypy . --strict

# All checks
black . && ruff check . && mypy . --strict
```

Environment Variables

```
# Telegram Configuration
TELEGRAM_API_ID=***
TELEGRAM_API_HASH=***
TELEGRAM_PHONE=***
TELEGRAM_PASSWORD=***

# API Keys
OPENAI_API_KEY=***
ANTHROPIC_API_KEY=***
PERPLEXITY_API_KEY=***

# Database
DATABASE_PATH=data/ai_rebuild.db
DATABASE_BACKUP_ON_STARTUP=true

# Monitoring
MONITORING_ENABLED=true
MONITORING_DASHBOARD_PORT=8080
```

Project Structure

```
ai/
|-- agents/
|   |-- valor/
|   |   |-- agent.py          # Main ValorAgent
|   |   +-- persona.md        # Persona definition
|   +-- subagents/            # Specialized subagents
|-- config/
|   |-- workspace_config.json # Multi-workspace config
|   +-- telegram_groups.json  # Group behavior config
|-- data/                     # SQLite databases
|-- docs/                     # Documentation
|-- integrations/
|   +-- telegram/             # Telegram handlers
|-- logs/                     # Log files
|-- mcp_servers/              # MCP tool servers
|-- scripts/                  # Operational scripts
|-- tests/                    # Test suites
|-- tools/                    # Tool implementations
+-- utilities/                # Shared utilities
```

Quick Reference

Useful Commands

Command	Description
`./scripts/start.sh`	Start the system
`./scripts/stop.sh`	Stop the system
`./scripts/logs.sh`	View logs
`./scripts/telegram_login.sh`	Authenticate Telegram
`curl localhost:9000/health`	Check health
`curl localhost:9000/resources/status`	Check resources

Key Endpoints

Endpoint	Description
`/health`	Basic health check
`/resources/status`	Detailed resource status
`/telegram/status`	Telegram connection status
`/daydreams`	View daydream insights
`/restart/status`	Auto-restart status

Critical Thresholds

Metric	Warning	Critical
Memory	600MB	800MB
CPU	80%	95%
Health Score	<70	<60
Sessions	80	100

Emergency Contacts

- **System Issues**: Check logs at `logs/system.log`
- **Telegram Issues**: Re-authenticate with `telegram_login.sh`
- **Database Issues**: Run startup script (includes recovery)

This documentation consolidates the complete Valor AI System documentation for easy reference and printing.

Document Version: 2.1 | Generated: December 17, 2025