Zen MCP Server: AI Orchestration Hub for Claude Code

> **Transform Claude into an AI development team orchestrator with multi-model collaboration, workflow-driven analysis, and conversation threading across specialized AI tools.**

Overview

Zen MCP Server is a **Model Context Protocol (MCP) server** that acts as an intelligent AI orchestration hub, enabling Claude to delegate complex tasks to specialized AI models while maintaining conversation context across different tools and workflows. Think of it as giving Claude the ability to consult with a team of AI specialists - each with their own expertise - while Claude remains the primary coordinator and executor.

Why Zen MCP Server?

The Problem: Claude is brilliant, but sometimes you need: - Multiple AI perspectives on complex decisions - Specialized models for specific tasks (Gemini for deep analysis, O3 for logical reasoning, GPT-5 for advanced capabilities) - Systematic workflows that prevent rushed analysis - Context that persists across tool switches and conversation resets

The Solution: Zen MCP Server creates a collaborative AI ecosystem where: - **Claude orchestrates** the workflow and performs actual implementation - **Specialized AI models** provide expert analysis and validation - **Conversation threads** maintain context across tool switches - **Workflow tools** enforce systematic investigation phases

Core Features

🧠 Multi-Model AI Orchestration

- **Intelligent Model Selection**: Auto-mode picks optimal models for each task, or specify models directly
- **Conversation Threading**: Context persists across tools and models with 1-hour memory using Redis
- **Collaborative Analysis**: Models can request follow-up from Claude and build on each other’s insights
- **Cost Optimization**: Smart token management and confidence-based expert consultation

🔧 Specialized Workflow Tools

- **Investigation-Driven**: Tools like `debug`, `codereview`, `analyze` enforce step-by-step investigation
- **Expert Validation**: High-confidence findings skip external validation to save costs
- **File-Aware Processing**: Intelligent file handling with token budget management
- **Multi-Repository Support**: Works across complex project structures

🌐 Extensive Model Support

- **Native APIs**: Gemini 2.5 Pro/Flash, OpenAI GPT-5/O3/O4, X.AI Grok models
- **Aggregated APIs**: OpenRouter (40+ models), DIAL platform (enterprise-grade)
- **Local Models**: Ollama, vLLM, LM Studio support for privacy and cost control
- **Custom Endpoints**: Any OpenAI-compatible API integration

📋 Advanced Capabilities

- **Vision Support**: Analyze images, diagrams, and screenshots with vision-capable models
- **Extended Context**: Bypass MCP's 25K token limit with model-specific handling
- **Context Revival**: Continue conversations after Claude's context resets
- **Structured Prompts**: Quick tool access via `/zen:tool` syntax in Claude Code

Installation

Prerequisites

- **Python 3.10+** (3.12 recommended)
- **Git** for cloning the repository
- **API Keys** (at least one):
  - Google AI Studio for Gemini models
  - OpenAI Platform for GPT-5/O3/O4 models
  - X.AI Console for Grok models
  - OpenRouter for multi-model access
  - DIAL Platform for enterprise AI orchestration
- **Windows Users**: WSL2 required for Claude Code CLI

Quick Installation (Recommended)

Option A: Zero-Setup with uvx

Install uv first: Follow uv installation guide

For Claude Code CLI - Create `.mcp.json` in your project root:

```
{
  "mcpServers": {
    "zen": {
      "command": "sh",
      "args": [
        "-c",
        "exec $(which uvx || echo uvx) --from git+https://
        github.com/BeehiveInnovations/zen-mcp-server.git zen-mcp-
        server"
      ],
      "env": {
        "PATH": "/usr/local/bin:/usr/bin:/bin:/opt/homebrew/
        bin:~/.local/bin",
        "GEMINI_API_KEY": "your_gemini_key_here",
        "OPENAI_API_KEY": "your_openai_key_here"
      }
    }
  }
}
```

For Claude Desktop - Add to Settings → Developer → Edit Config:

```
{
  "mcpServers": {
    "zen": {
      "command": "sh",
      "args": [
        "-c",
        "exec $(which uvx || echo uvx) --from git+https://
        github.com/BeehiveInnovations/zen-mcp-server.git zen-mcp-
        server"
      ],
      "env": {
        "PATH": "/usr/local/bin:/usr/bin:/bin:/opt/homebrew/
        bin:~/.local/bin",
        "GEMINI_API_KEY": "your_gemini_key_here",
        "OPENAI_API_KEY": "your_openai_key_here"
      }
    }
  }
}
```

Option B: Traditional Setup

```
# Clone the repository
git clone https://github.com/BeehiveInnovations/zen-mcp-
        server.git
cd zen-mcp-server

# Automated setup (installs dependencies and configures Claude)
./run-server.sh

# Windows PowerShell users:
./run-server.ps1

# View configuration for manual setup
./run-server.sh -c
```

Environment Configuration

Create or edit `.env` file in the project root:

```
# === CORE API KEYS (at least one required) ===
GEMINI_API_KEY=your_gemini_api_key_here
OPENAI_API_KEY=your_openai_api_key_here
XAI_API_KEY=your_xai_api_key_here

# === AGGREGATED PROVIDERS ===
OPENROUTER_API_KEY=your_openrouter_key_here
DIAL_API_KEY=your_dial_api_key_here

# === LOCAL/CUSTOM MODELS ===
CUSTOM_API_URL=http://localhost:11434/v1  # Ollama example
CUSTOM_API_KEY=                           # Leave empty for
        Ollama
CUSTOM_MODEL_NAME=llama3.2               # Default model name

# === CONFIGURATION ===
DEFAULT_MODEL=auto                        # Auto-select best
        model per task
REDIS_URL=redis://localhost:6379          # Conversation memory
        (optional)
WORKSPACE_ROOT=/path/to/your/projects     # File access root
        (defaults to $HOME)
LOG_LEVEL=INFO                           # Logging verbosity
```

No restart needed - Changes to `.env` take effect immediately.

Verification

Test your installation:

```
# Start Claude Code CLI in your project directory
claude

# Test with a simple query
ask "Use zen to list available models"

# Test a workflow tool
ask "Use zen debug to help me understand why my test is failing"
```

Tools and Routing Overview

Zen MCP Server provides 16 specialized tools organized into three categories: **Simple Tools**, **Workflow Tools**, and **Utility Tools**. Each tool is designed for specific development scenarios and automatically routes to optimal AI models based on task requirements.

Model Routing Strategy

Auto Mode (DEFAULT_MODEL=auto): Claude intelligently selects models based on tool requirements:

🧠 Extended Reasoning Tasks

- **Tools**: `thinkdeep`, `debug`, `secaudit`, `analyze`, `consensus`
- **Routing Priority**: GPT-5 → Grok-4-Heavy → O3 → Grok-4 → Gemini-2.5-Pro
- **Use Cases**: Complex analysis, security audits, architectural decisions

⚡ Fast Response Tasks

- **Tools**: `chat`, `challenge`, `listmodels`, `version`
- **Routing Priority**: GPT-5-Nano → O4-mini → GPT-5-Mini → O3-mini → Grok-3-Fast
- **Use Cases**: Quick consultation, validation, system information

⚖️ Balanced Tasks

- **Tools**: `planner`, `codereview`, `precommit`, `refactor`, `tracer`, `testgen`, `docgen`
- **Routing Priority**: GPT-5-Mini → O4-mini → O3-mini → Grok-4 → Gemini-2.5-Flash
- **Use Cases**: Structured workflows, code generation, documentation

Tool Categories

Simple Tools (Direct AI Consultation)

`chat` - Collaborative Development Partner - **Purpose**: Brainstorming, architecture discussions, second opinions - **Model Selection**: Fast response models for real-time collaboration - **Features**: Web search, continuation support, thinking modes - **Example**: `"Chat with zen about the best authentication strategy for my React app"`

`challenge` - Critical Analysis Enforcer
- **Purpose**: Prevents reflexive agreement, encourages critical thinking - **Model Selection**: Reasoning-focused models for objective analysis - **Features**: Anti-bias prompting, assumption challenging - **Example**: `"Challenge: isn't adding this method to the base class a bad idea?"`

`thinkdeep` - Extended Reasoning Partner - **Purpose**: Deep analysis, edge case identification, alternative perspectives - **Model Selection**: Extended reasoning models with thinking capabilities - **Features**: Multi-step investigation, hypothesis testing, confidence tracking - **Example**: `"Use thinkdeep to analyze why this algorithm fails on large datasets"`

`consensus` - Multi-Model Decision Analysis - **Purpose**: Get diverse expert opinions on technical decisions - **Model Selection**: Multiple models with stance configuration (for/against/neutral) - **Features**: Structured debate, decision matrix, expert validation - **Example**: `"Get consensus with flash:for and o3:against on migrating to GraphQL"`

Workflow Tools (Systematic Investigation)

These tools enforce structured investigation phases with mandatory pauses between steps, ensuring thorough analysis before expert consultation.

`debug` - Systematic Bug Investigation - **Process**: Issue description → Code examination → Evidence collection → Hypothesis formation → Expert validation - **Investigation Steps**: 1. Problem statement and initial investigation plan 2. Code analysis and evidence gathering

(MANDATORY investigation between calls) 3. Root cause identification and solution recommendation - **Confidence Tracking**: `exploring` → `low` → `medium` → `high` → `certain` - **Expert Bypass**: When confidence reaches `certain`, skips external model consultation - **Example**: `"Debug why the sync process gets stuck without errors in diagnostics.log"`

`codereview` - Professional Code Audit - **Process**: Review scope → Code examination → Issue identification → Severity assessment → Expert analysis - **Investigation Steps**: 1. Review plan and file identification 2. Code quality analysis (MANDATORY investigation between calls) 3. Security and performance assessment with final recommendations - **Issue Classification**: Critical → High → Medium → Low severity levels - **Pattern Detection**: Code smells, anti-patterns, architectural concerns - **Example**: `"Perform codereview with gemini pro focusing on auth.py security vulnerabilities"`

`precommit` - Pre-Commit Validation - **Process**: Git analysis → Change examination → Impact assessment → Regression detection → Validation - **Investigation Steps**: 1. Git status and change scope analysis 2. Diff examination and impact assessment (MANDATORY investigation between calls) 3. Regression and compatibility validation with commit recommendation - **Multi-Repository**: Supports complex project structures - **Change Types**: Staged, unstaged, specific branch comparisons - **Example**: `"Precommit validation with o3 to ensure no regressions in payment processing"`

`analyze` - Comprehensive Code Analysis - **Process**: Analysis scope → Structure examination → Pattern identification → Architecture assessment → Strategic recommendations - **Investigation Steps**: 1. Analysis plan and file examination strategy 2. Code structure and pattern analysis (MANDATORY investigation between calls) 3. Architecture assessment with improvement opportunities - **Analysis Types**: Architecture, performance, security, quality, general - **Output Formats**: Summary, detailed, actionable recommendations - **Example**: `"Analyze the microservices architecture for scalability bottlenecks"`

`refactor` - Intelligent Code Restructuring - **Process**: Refactor scope → Code smell detection → Decomposition analysis → Modernization opportunities → Implementation plan - **Investigation Steps**: 1. Refactoring objectives and code examination 2. Code smell and decomposition opportunity analysis (MANDATORY investigation between calls) 3. Modernization recommendations with implementation guidance - **Refactor Types**: Code smells, decomposition, modernization, organization - **Focus Areas**: Performance, readability, maintainability, security - **Example**: `"Refactor the UserManager class with decomposition focus using gemini pro"`

`testgen` - Comprehensive Test Generation - **Process**: Test scope → Code analysis → Edge case identification → Test framework detection → Test generation - **Investigation Steps**: 1. Test requirements and code functionality analysis 2. Critical path and edge case identification (MANDATORY investigation between calls) 3. Comprehensive test suite generation with framework integration - **Coverage Types**: Unit, integration, edge cases, error conditions - **Framework Detection**: Automatic test pattern following - **Example**: `"Generate comprehensive tests for User.login() with edge case coverage"`

`secaudit` - Security Audit Framework - **Process**: Security scope → Vulnerability scan → OWASP analysis → Compliance check → Risk assessment - **Investigation Steps**: 1. Security audit plan and threat model identification 2. OWASP Top 10 analysis and vulnerability assessment (MANDATORY investigation between calls) 3. Compliance evaluation with remediation recommendations - **Audit Focus**: OWASP, compliance (SOC2, PCI DSS, HIPAA, GDPR), infrastructure, dependencies - **Threat Levels**: Low, medium, high, critical risk classifications - **Example**: `"Security audit with o3 focusing on payment processing PCI DSS compliance"`

`docgen` - Documentation Generation - **Process**: Documentation scope → Code analysis → Complexity assessment → Documentation generation → Quality review - **Investigation Steps**: 1. Documentation requirements and code structure analysis 2. Function complexity and dependency analysis (MANDATORY investigation between calls)
3. Comprehensive documentation generation with gotcha identification - **Features**: Big-O complexity notation, call flow documentation, gotcha detection - **Documentation Types**: Function docs, class documentation, complexity analysis - **Example**: `"Generate documentation for UserManager with complexity analysis and gotchas"`

Utility Tools

`planner` - Interactive Project Planning - **Features**: Step-by-step planning, branch exploration, revision capabilities - **Planning Types**: Feature development, migration strategies, architectural decisions - **Example**: `"Plan the microservices migration with dependency analysis"`

`tracer` - Code Flow Analysis - **Modes**: Precision (execution flow), Dependencies (structural relationships) - **Analysis Types**: Call chains, usage patterns, dependency mapping - **Example**: `"Trace how UserAuthManager.authenticate flows through the system"`

`listmodels` - Model Information - **Features**: Provider status, model capabilities, configuration details - **Information**: Context windows, vision support, thinking modes - **Example**: `"List all available models with their capabilities"`

`version` - Server Diagnostics - **Features**: Version information, configuration status, system health - **Diagnostics**: Provider connectivity, Redis status, environment validation - **Example**: `"Check zen server version and configuration"`

Workflow Investigation Enforcement

Key Innovation: Workflow tools enforce systematic investigation through:

1. **Mandatory Investigation Phases**: After each tool call, Claude MUST examine code/files before calling the tool again
2. **Evidence-Based Progression**: Each step requires NEW evidence from actual code analysis
3. **No Recursive Calls**: Tools reject immediate re-calls without investigation work
4. **Required Actions Lists**: Tools provide specific investigation guidance
5. **Confidence-Based Expert Consultation**: High confidence findings can skip external validation

Example Debug Workflow:

```
Step 1: "I need to debug why sync gets stuck"
→ Tool Response: "MANDATORY: Examine sync code and logs before
Step 2"
→ Claude investigates: Reads sync.py, analyzes diagnostics.log,
examines related files
Step 2: "Found deadlock in thread pool, confidence: high"
→ Tool Response: "Root cause identified with high confidence,
providing solution..."
```

File Processing and Token Management

Intelligent File Handling: - **Automatic Directory Expansion**: Recursively discovers relevant files - **Token Budget Management**: Respects model-specific context limits - **Conversation-Aware**: Avoids re-processing files already in conversation - **Large File Support**: Uses MCP's file mechanism for oversized content

Token Allocation by Model: - **Gemini 2.5 Pro/Flash**: 1M context → 800K for files, 200K for conversation - **GPT-5 Family**: 400K context → 320K for files, 80K for conversation - **O3/O4 Models**: 200K context → 160K for files, 40K for conversation - **Custom Models**: Configurable limits in `conf/custom_models.json`

Important Information

Conversation Threading and Memory

Redis-Based Threading: Conversations persist across tool switches with 1-hour expiry - **Thread Continuity**: Models receive full conversation history - **Cross-Tool Context**: Switch between tools while maintaining context - **Context Revival**: Continue conversations after Claude's context resets - **Memory Management**: Automatic cleanup and optimization

Configuration:

```
REDIS_URL=redis://localhost:6379  # Local Redis
# Or use cloud Redis for persistence across sessions
REDIS_URL=redis://your-cloud-redis-url:6379
```

Cost Optimization Features

Confidence-Based Expert Consultation: - **High Confidence Skip**: When Claude reaches `certain` confidence, skips external model calls - **Forced Consultation**: Use `must debug using model_name` to force expert validation - **Local-Only Mode**: Add `do not use another model` to run workflows locally - **Smart Token Management**: Optimizes file processing based on model capabilities

Model Cost Tiers: - **Premium**: O3-Pro (use sparingly), GPT-5 (with reasoning_effort) - **Balanced**: GPT-5-Mini, Grok-4, Gemini-2.5-Pro - **Economical**: GPT-5-Nano, O4-mini, Gemini-2.5-Flash, local models

Security and Privacy

API Key Security: - **Environment Variables**: Never hardcode keys in configuration files - **Local Models**: Use Ollama, vLLM for sensitive projects requiring privacy - **Access Control**: Configure `WORKSPACE_ROOT` to limit file access scope

Data Handling: - **No Persistent Storage**: File content is processed in-memory only - **Conversation Memory**: Optional Redis storage with automatic expiry - **Model Isolation**: Each conversation thread is independent

Performance Optimization

Model Selection Strategy:

```
# Auto mode for intelligent selection
DEFAULT_MODEL=auto

# Or set specific defaults
DEFAULT_MODEL=gemini-2.5-flash  # Fast responses
DEFAULT_MODEL=o3                # Strong reasoning
DEFAULT_MODEL=gpt-5             # Advanced capabilities
```

Thinking Mode Configuration:

```
# Control thinking depth vs token cost
# minimal (0.5%) → low (8%) → medium (33%) → high (67%) → max
(100%)
DEFAULT_THINKING_MODE=medium
```

Conversation Settings:

```
CONVERSATION_TIMEOUT=3600        # 1 hour conversation memory
MAX_CONVERSATION_TURNS=20        # Turn limit per thread
```

Advanced Configuration

Custom Model Integration: Edit `conf/custom_models.json` for OpenRouter, DIAL, or local models:

```json
{
  "providers": {
    "openrouter": {
      "base_url": "https://openrouter.ai/api/v1",
      "api_key_env": "OPENROUTER_API_KEY",
      "models": {
        "claude-opus": {
          "name": "anthropic/claude-3-5-sonnet-20241022",
          "context_window": 200000,
          "supports_vision": true
        }
      }
    }
  }
}
```

Logging Configuration:

```
LOG_LEVEL=INFO          # DEBUG, INFO, WARNING, ERROR
LOG_FORMAT=structured   # structured, simple
LOG_TO_FILE=true       # Enable file logging
```

Troubleshooting

Common Issues

1. "No models available" Error

Symptoms: Tools report no models configured or API key errors

Solutions:

```
# Check API key configuration
grep -E "(GEMINI|OPENAI|XAI)_API_KEY" .env

# Test API connectivity
curl -H "Authorization: Bearer $OPENAI_API_KEY" \
     https://api.openai.com/v1/models

# Verify model configuration
ask "Use zen to list available models"
```

Common Causes: - Missing or invalid API keys - Network connectivity issues - Provider service outages - Insufficient API quota/credits

2. Workflow Tools Not Responding

Symptoms: Debug, codereview, or analyze tools hang or timeout

Solutions:

```
# Check Redis connectivity (if using conversation memory)
redis-cli ping

# Verify file access permissions
ls -la /path/to/your/project

# Test with smaller file set
ask "Use zen debug with just this one file: main.py"
```

Common Causes: - Redis connection issues - File permission restrictions - Large file processing timeout - Network latency with cloud Redis

3. Model Selection Issues

Symptoms: Wrong model selected or specific model requests failing

Solutions:

```
# Force specific model selection
ask "Use zen chat with o3 specifically to analyze this code"

# Check model aliases
```

```
cat conf/custom_models.json

# Verify provider configuration
ask "Use zen version to check configuration"
```

Common Causes: - Model name conflicts between providers - Provider-specific model availability - Custom model configuration errors

4. Context/Memory Issues

Symptoms: Lost conversation context or repeated analysis

Solutions:

```
# Check Redis status
redis-cli info memory

# Restart conversation thread
ask "Start a new zen conversation thread"

# Verify conversation limits
grep -E "(CONVERSATION|TIMEOUT)" .env
```

Common Causes: - Redis memory limits exceeded - Conversation timeout reached - Network interruption with cloud Redis

5. File Processing Errors

Symptoms: “Could not read file” or token limit exceeded errors

Solutions:

```
# Check file permissions
chmod 644 /path/to/files

# Verify workspace root configuration
echo $WORKSPACE_ROOT

# Use file references instead of full content
ask "Use zen analyze with file references only"
```

Common Causes: - File permission restrictions - Files outside workspace root - Binary files included in processing - Model context window exceeded

Diagnostic Commands

System Health Check:

```
# Check server status
ask "Use zen version"
```

```
# Test model connectivity
ask "Use zen listmodels"

# Verify file access
ask "Use zen chat to tell me what files you can see in this
        directory"
```

Configuration Validation:

```
# Display current configuration
./run-server.sh -c

# Check environment variables
env | grep -E "(GEMINI|OPENAI|XAI|REDIS)"

# Test API endpoints
curl -I https://generativelanguage.googleapis.com/
curl -I https://api.openai.com/
```

Performance Monitoring:

```
# Redis memory usage
redis-cli info memory

# Check conversation threads
redis-cli keys "zen:conversation:*"

# Monitor API usage
tail -f logs/zen-mcp-server.log
```

Advanced Debugging

Enable Debug Logging:

```
# Edit .env file
LOG_LEVEL=DEBUG
LOG_TO_FILE=true

# Check detailed logs
tail -f logs/zen-mcp-server.log
```

Network Diagnostics:

```
# Test API endpoints
curl -H "Authorization: Bearer $OPENAI_API_KEY" \
     https://api.openai.com/v1/models

curl -H "x-goog-api-key: $GEMINI_API_KEY" \
     https://generativelanguage.googleapis.com/v1beta/models
```

Redis Connection Testing:

```
# Local Redis
redis-cli ping

# Remote Redis
redis-cli -u $REDIS_URL ping

# Check connection info
redis-cli -u $REDIS_URL info server
```

Getting Help

Documentation Resources: - Advanced Usage Guide - Detailed configuration and parameters - Custom Models Setup - OpenRouter, DIAL, and local model configuration - WSL Setup Guide - Windows-specific installation instructions - Tool Documentation - Individual tool guides and examples

Community Support: - GitHub Issues: https://github.com/BeehiveInnovations/zen-mcp-server/issues - Discussions: https://github.com/BeehiveInnovations/zen-mcp-server/discussions

Quick Support Template:

```
**Environment:**
- OS: [macOS/Linux/Windows+WSL]
- Python Version: [run `python --version`]
- Claude Client: [Code CLI/Desktop/Gemini CLI]

**Configuration:**
- API Providers: [Gemini/OpenAI/XAI/OpenRouter/Custom]
- Default Model: [auto/specific model]
- Redis: [local/cloud/disabled]

**Issue:**
[Describe the problem with specific error messages]

**Steps to Reproduce:**
1. [First step]
2. [Second step]
3. [Error occurs]

**Expected vs Actual:**
- Expected: [What should happen]
- Actual: [What actually happens]
```

Built with ❤️ for the Claude Code and AI development community