AI Operating System - 2GB RAM + FREE Tier Implementation

Complete MCP-Based Architecture Using Hugging Face API

> **Perfect for Students, Hackathons, Personal Projects, and Testing**
> Build a fully functional AI Operating System with just 2GB RAM and $0 monthly cost using Hugging Face Inference API

🎯 OVERVIEW

This guide provides a complete implementation of an AI Operating System using the MCP (Model Context Protocol) architecture, optimized to run on just 2GB RAM with completely free API access through Hugging Face. The system leverages GPT-OSS models via Hugging Face Inference API and provides natural language control over your computer.

Core Capabilities

- **Natural Language Commands**: "Open WhatsApp and call Kartik", "Create a Python project", "Schedule a meeting"
- **System Operations**: File management, application launching, process control
- **Development Tools**: VS Code integration, Git operations, code analysis
- **Communication**: WhatsApp automation, email management
- **Web Integration**: Browser automation, search capabilities, content extraction
- **AI Processing**: Complex reasoning via Hugging Face GPT-OSS API

Resource Requirements

- **RAM**: 2GB minimum (4GB recommended)
- **Storage**: 3GB minimum
- **CPU**: Any dual-core processor
- **Cost**: $0/month (using Hugging Face free tier)
- **Internet**: Required for Hugging Face API calls

AI Model Access Strategy

- **Primary**: Hugging Face Inference API for GPT-OSS models
- **Backup**: Groq free tier (14,400 tokens/day)
- **Emergency**: Gemini free tier (1,500 requests/day)
- **Local Fallback**: Ollama (slower but unlimited)

🏗️ COMPLETE MONOREPO STRUCTURE

```
ai-os-monorepo/
├── README.md                    # Main documentation and setup guide
├── LICENSE                      # Apache 2.0 license file
├── docker-compose.yml           # Container orchestration for all MCP servers
├── .env.example                 # Environment variables template with API keys
├── package.json                 # Node.js workspace configuration
├── pyproject.toml               # Python dependencies and MCP package config
├── requirements.txt             # Core Python requirements including mcp[cli]
├── .gitignore                   # Git ignore patterns
│
├── core/                        # Core system components
│   ├── __init__.py
│   ├── orchestrator/            # Central MCP Orchestrator (Port 9000)
│   │   ├── __init__.py
│   │   ├── src/
│   │   │   ├── main.py          # FastMCP orchestrator server - routes commands to appropriate MCP servers
│   │   │   ├── router.py        # CommandRouter class - parses natural language and determines server routing
│   │   │   ├── session.py       # SessionManager class - handles multi-user sessions and context
```

```
│   │   │   ├── config.py          # Configuration class - manages MCP server connections and settings
│   │   │   ├── health.py          # HealthMonitor class - checks MCP server status and availability
│   │   │   └── middleware.py      # RequestMiddleware class - handles authentication and rate limiting
│   │   ├── requirements.txt       # Dependencies: mcp[cli], fastapi, uvicorn
│   │   ├── Dockerfile             # Lightweight container configuration
│   │   └── README.md              # Orchestrator setup and usage documentation
│   │
│   ├── shared/                    # Shared utilities and libraries
│   │   ├── __init__.py
│   │   ├── harmony_client.py      # HarmonyClient class - handles Harmony format for GPT-OSS
│   │   ├── huggingface_api.py     # HuggingFaceClient class - manages GPT-OSS API calls via Hugging Face
│   │   ├── cloud_api.py           # CloudAPIManager class - handles multiple API providers with fallbacks
│   │   ├── logging.py             # LoggingService class - unified logging across all MCP servers
│   │   ├── types.py               # Common type definitions for MCP messages and responses
│   │   ├── utils.py               # Helper functions for command parsing and validation
│   │   ├── security.py            # SecurityManager class - handles API key management and validation
│   │   └── exceptions.py          # Custom exception classes for MCP errors
│   │
│   └── config/                    # Configuration files
│       ├── default.json           # Default MCP server configurations and ports
│       ├── development.json       # Development settings with debug modes
```

```
│       ├── production.json          # Production settings with optimizations
│       └── free_tier.json           # Free tier API limits and usage tracking
│
├── servers/                         # MCP Servers Implementation
│   ├── available/                   # ✅ READY-TO-USE SERVERS
│   │   ├── browser/                 # Browser automation MCP server (Port 8001)
│   │   │   ├── server.py            # BrowserMCPServer - search, open, find tools using Exa backend
│   │   │   ├── requirements.txt     # Dependencies: mcp[cli], selenium, exa-py
│   │   │   ├── Dockerfile           # Browser server container
│   │   │   └── README.md            # Browser server documentation
│   │   │
│   │   ├── python/                  # Python execution MCP server (Port 8000)
│   │   │   ├── server.py            # PythonMCPServer - stateless code execution in Docker containers
│   │   │   ├── requirements.txt     # Dependencies: mcp[cli], docker, python-docker
│   │   │   ├── Dockerfile           # Python execution server container
│   │   │   └── README.md            # Python server documentation
│   │   │
│   │   └── gpt_oss_mcp/             # GPT-OSS MCP integration from repository
│   │       ├── browser_server.py    # BrowserMCPAdapter - adapts GPT-OSS browser tool to MCP
│   │       ├── python_server.py     # PythonMCPAdapter - adapts GPT-OSS python tool to MCP
│   │       ├── build-system-prompt.py # SystemPromptBuilder - generates system prompts via MCP discovery
│   │       ├── reference-system-prompt.py # ReferencePromptGenerator - creates reference system prompts
```

```
│   │       ├── pyproject.toml       # MCP integration package configuration
│   │       └── README.md            # GPT-OSS MCP integration documentation
│   │
│   ├── marketplace/                 # 🔌 AUTO-INSTALL FROM MCP MARKETPLACE
│   │   ├── install_manager.py       # MarketplaceInstaller class - automates MCP server installation
│   │   ├── available_servers.json   # ServerCatalog - list of available MCP servers from marketplace
│   │   ├── install_scripts/         # Installation automation scripts
│   │   │   ├── github.sh            # install_github_server() - GitHub API integration MCP server
│   │   │   ├── git.sh               # install_git_server() - Git operations MCP server
│   │   │   ├── filesystem.sh        # install_filesystem_server() - File system operations MCP server
│   │   │   ├── slack.sh             # install_slack_server() - Slack messaging integration MCP server
│   │   │   ├── docker.sh            # install_docker_server() - Docker container management MCP server
│   │   │   ├── gmail.sh             # install_gmail_server() - Gmail API integration MCP server
│   │   │   ├── calendar.sh          # install_calendar_server() - Calendar operations MCP server
│   │   │   ├── notion.sh            # install_notion_server() - Notion database access MCP server
│   │   │   └── database.sh          # install_database_server() - Database connection MCP server
│   │   │
│   │   └── configs/                 # MCP server configuration templates
│   │       ├── github_config.json   # GitHub server configuration with API keys and permissions
│   │       ├── git_config.json      # Git server configuration with repository settings
│   │       ├── filesystem_config.json # Filesystem server configuration with access permissions
```

```
│   │       ├── slack_config.json    # Slack server configuration with bot tokens and channels
│   │       ├── docker_config.json   # Docker server configuration with container settings
│   │       ├── gmail_config.json    # Gmail server configuration with OAuth and API settings
│   │       ├── calendar_config.json # Calendar server configuration with API credentials
│   │       ├── notion_config.json   # Notion server configuration with database IDs and tokens
│   │       └── database_config.json # Database server configuration with connection strings
│   │
│   └── custom/                    # 🔨 BUILD OURSELVES (OPTIMIZED FOR 2GB)
│       ├── system_ops/            # System operations MCP server (Port 8002)
│       │   ├── __init__.py
│       │   ├── src/
│       │   │   ├── main.py        # SystemOpsMCPServer - FastMCP server for system operations
│       │   │   ├── app_launcher.py  # AppLauncher class - cross-platform application launching
│       │   │   ├── file_ops.py     # FileOperations class - lightweight file management
│       │   │   ├── process_manager.py # ProcessManager class - basic process monitoring and control
│       │   │   ├── system_info.py  # SystemInfo class - hardware and OS information gathering
│       │   │   └── hardware.py     # HardwareInterface class - basic hardware interaction
│       │   ├── requirements.txt    # Dependencies: mcp[cli], psutil, pathlib
│       │   ├── Dockerfile          # Optimized lightweight container
│       │   └── README.md           # System operations server documentation
│       │
│       ├── communication/          # Communication automation MCP server (Port 8003)
│       │   ├── __init__.py
```

```
│       │   ├── src/
│       │   │   ├── main.py        # CommunicationMCPServer - FastMCP server for messaging
│       │   │   ├── whatsapp_web.py # WhatsAppWebAutomation class - WhatsApp Web automation via Selenium
│       │   │   ├── phone_calls.py  # PhoneCallManager class - system-dependent phone integration
│       │   │   ├── email_client.py # EmailClient class - Gmail API integration for email management
│       │   │   ├── sms_handler.py  # SMSHandler class - SMS sending and receiving
│       │   │   └── social_media.py # SocialMediaManager class - automated posting to platforms
│       │   ├── requirements.txt    # Dependencies: mcp[cli], selenium, google-api-python-client
│       │   ├── Dockerfile          # Communication server container
│       │   └── README.md           # Communication server documentation
│       │
│       └── ide_integration/        # IDE control MCP server (Port 8004)
│           ├── __init__.py
│           ├── src/
│           │   ├── main.py        # IDEIntegrationMCPServer - FastMCP server for IDE control
│           │   ├── vscode_api.py   # VSCodeClient class - VS Code API integration and control
│           │   ├── file_editor.py  # FileEditor class - direct file editing and manipulation
│           │   ├── git_ops.py      # GitOperations class - Git command wrapper and automation
│           │   ├── code_analysis.py # CodeAnalyzer class - basic code quality analysis
│           │   └── project_manager.py # ProjectManager class - project creation and management
│           ├── requirements.txt    # Dependencies: mcp[cli], pygit2, python-language-server
│           ├── Dockerfile          # IDE integration server container
```

```
│           └── README.md          # IDE integration server documentation
│
├── frontend/                      # User interfaces
│   ├── web/                       # Lightweight web dashboard
│   │   ├── src/
│   │   │   ├── components/        # React components for UI
│   │   │   │   ├── CommandInput.tsx # CommandInput component - natural language command interface
│   │   │   │   ├── ChatInterface.tsx # ChatInterface component - conversation-style interaction
│   │   │   │   ├── SystemStatus.tsx # SystemStatus component - MCP server health monitoring
│   │   │   │   └── ToolsPanel.tsx  # ToolsPanel component - available MCP tools display
│   │   │   ├── services/          # API service clients
│   │   │   │   ├── orchestrator.ts # OrchestratorClient - connects to central MCP orchestrator
│   │   │   │   ├── commands.ts     # CommandService - handles command execution and responses
│   │   │   │   └── websocket.ts    # WebSocketService - real-time communication with MCP servers
│   │   │   ├── styles/            # Minimal styling
│   │   │   │   └── globals.css     # Global CSS styles for lightweight UI
│   │   │   ├── App.tsx            # Main React application component
│   │   │   └── index.tsx          # Application entry point
│   │   ├── package.json           # Frontend dependencies: react, typescript, axios
│   │   ├── Dockerfile             # Web interface container
│   │   └── README.md              # Web interface documentation
│   │
```

```
│   ├── cli/                    # Command line interface
│   │   ├── main.py             # CLIInterface class - command line entry point for AI OS
│   │   ├── commands.py         # CommandHandler class - CLI command processing and routing
│   │   ├── interactive.py      # InteractiveMode class - interactive command session
│   │   └── README.md           # CLI interface documentation
│   │
│   └── desktop/                # Optional desktop application
│       ├── src-tauri/          # Tauri backend for native desktop app
│       ├── src/                # Frontend code for desktop interface
│       ├── tauri.conf.json     # Tauri configuration for desktop app
│       └── README.md           # Desktop app documentation
│
├── deployment/                 # Deployment configurations
│   ├── docker/
│   │   ├── docker-compose.local.yml    # docker_compose_local() - local development setup
│   │   ├── docker-compose.2gb.yml      # docker_compose_2gb() - 2GB RAM optimized configuration
│   │   ├── docker-compose.prod.yml     # docker_compose_production() - production deployment setup
│   │   ├── Dockerfile.base             # Base Docker image with common dependencies
│   │   └── healthcheck.sh              # health_check_script() - container health verification
│   │
│   ├── kubernetes/                     # Kubernetes manifests (optional)
│   │   ├── namespace.yaml              # kubernetes_namespace() - isolated environment setup
│   │   ├── orchestrator.yaml           # orchestrator_deployment() - central MCP server deployment
```

```
│   │   ├── configmap.yaml         # configuration_map() - shared configuration across pods
│   │   └── services.yaml          # service_definitions() - network access to MCP servers
│   │
│   └── scripts/                   # Deployment automation scripts
│       ├── setup.sh               # one_click_setup() - complete system installation
│       ├── install_marketplace.sh     # install_marketplace_servers() - automated MCP server installation
│       ├── start_dev.sh           # start_development_mode() - development environment startup
│       ├── start_2gb.sh           # start_2gb_optimized() - memory-optimized startup
│       ├── health_check.sh        # system_health_check() - verify all MCP servers running
│       └── backup.sh              # backup_system_data() - backup configurations and data
│
├── tests/                         # Testing framework
│   ├── unit/                      # Unit tests for individual components
│   │   ├── test_orchestrator.py   # test_orchestrator_functionality() - test command routing
│   │   ├── test_system_ops.py     # test_system_operations() - test file and app operations
│   │   ├── test_communication.py  # test_communication_server() - test messaging functionality
│   │   └── test_ide_integration.py # test_ide_integration() - test VS Code and Git operations
│   ├── integration/               # Integration tests for MCP server communication
│   │   ├── test_full_workflow.py  # test_full_user_workflow() - end-to-end command execution
│   │   ├── test_api_endpoints.py  # test_api_endpoints() - test Hugging Face API integration
│   │   └── test_free_tier_limits.py # test_free_tier_usage() - verify API limit compliance
│   ├── e2e/                       # End-to-end user scenario tests
│   │   ├── test_user_scenarios.py  # test_real_user_scenarios() - test common user workflows
```

```
│   │   └── test_command_execution.py # test_command_execution() - test natural language processing
│   └── fixtures/                # Test data and mock responses
│       ├── sample_commands.json    # sample_command_data() - example commands for testing
│       └── mock_responses.json     # mock_api_responses() - simulated API responses
│
├── docs/                        # Documentation
│   ├── setup.md                 # Complete setup and installation instructions
│   ├── api.md                   # API documentation for all MCP servers
│   ├── architecture.md          # System architecture and MCP server interactions
│   ├── mcp_servers.md           # MCP server development and customization guide
│   ├── free_tier_guide.md       # Free tier optimization strategies and limits
│   ├── troubleshooting.md       # Common issues and solutions
│   └── examples/                # Usage examples and tutorials
│       ├── basic_commands.md      # Basic command examples and usage
│       ├── automation_workflows.md # Complex automation workflow examples
│       └── integration_examples.md # Third-party integration examples
│
├── examples/                    # Example implementations and demos
│   ├── basic_commands.py         # BasicCommandExamples class - simple command demonstrations
│   ├── automation_examples.py     # AutomationWorkflows class - complex task automation examples
│   ├── integration_demos.py       # IntegrationDemos class - third-party service integrations
│   └── free_tier_optimization.py  # FreeTierOptimizer class - API usage optimization patterns
```

```
│
└── tools/                    # Development and maintenance utilities
    ├── monitor.py              # SystemMonitor class - real-time MCP server monitoring
    ├── api_usage_tracker.py      # APIUsageTracker class - track Hugging Face API usage
    ├── performance_profiler.py    # PerformanceProfiler class - system performance analysis
    └── log_analyzer.py           # LogAnalyzer class - centralized log analysis and insights
```

🔧 DETAILED MCP SERVER ARCHITECTURE

MCP Server Network Layout

```
MCP_SERVER_ARCHITECTURE:
├── Central Orchestrator (Port 9000) 🔴 BUILD
│   ├── FastMCP server - main.py
│   ├── CommandRouter class - routes natural language to MCP servers
│   ├── SessionManager class - manages multi-user contexts
│   ├── HuggingFaceClient class - GPT-OSS API integration
│   └── HealthMonitor class - monitors all MCP server status
│
├── Browser Server (Port 8001) ✅ AVAILABLE
│   ├── BrowserMCPServer - browser_server.py from GPT-OSS repo
│   ├── search() tool - web search using Exa backend
│   ├── open() tool - page navigation and content extraction
│   ├── find() tool - content search within pages
│   └── CitationManager class - manages source citations
```

```
│
├── Python Server (Port 8000) ✅ AVAILABLE
│   ├── PythonMCPServer - python_server.py from GPT-OSS repo
│   ├── execute() tool - stateless Python code execution
│   ├── DockerContainer class - sandboxed execution environment
│   ├── OutputCapture class - captures stdout/stderr
│   └── SecuritySandbox class - prevents malicious code execution
│
├── System Operations Server (Port 8002) 🔴 BUILD
│   ├── SystemOpsMCPServer - main.py using FastMCP framework
│   ├── launch_app() tool - cross-platform application launching
│   ├── file_operations() tool - create, read, write, delete files
│   ├── list_processes() tool - running process monitoring
│   ├── system_info() tool - hardware and OS information
│   └── ProcessManager class - basic process control
│
├── Communication Server (Port 8003) 🔴 BUILD
│   ├── CommunicationMCPServer - main.py using FastMCP framework
│   ├── send_whatsapp() tool - WhatsApp Web message automation
│   ├── make_call() tool - phone call initiation
│   ├── send_email() tool - Gmail API email sending
│   ├── WhatsAppWebAutomation class - Selenium-based WhatsApp control
│   └── EmailClient class - Gmail API integration
│
```

```
└── IDE Integration Server (Port 8004) 🔴 BUILD
    ├── IDEIntegrationMCPServer - main.py using FastMCP framework
    ├── open_vscode() tool - VS Code project opening
    ├── create_project() tool - project scaffolding and setup
    ├── git_operations() tool - Git commit, push, pull automation
    ├── VSCodeClient class - VS Code API integration
    └── GitOperations class - Git command wrapper
```

Hugging Face Integration Strategy

```
HUGGING_FACE_API_INTEGRATION:
├── Primary Models (via Hugging Face Inference API)
│   ├── openai/gpt-oss-120b - complex reasoning and planning
│   ├── openai/gpt-oss-20b - faster responses and simple tasks
│   └── HuggingFaceClient class - API wrapper with rate limiting
│
├── Free Tier Management
│   ├── APIUsageTracker class - monitors daily/monthly limits
│   ├── RequestQueue class - queues requests during rate limits
│   ├── FallbackManager class - switches to backup APIs when needed
│   └── CacheManager class - caches responses to reduce API calls
│
├── Backup APIs (Free Tiers)
│   ├── Groq API - 14,400 tokens/day free
│   ├── Google Gemini - 1,500 requests/day free
```

```
│   ├── Anthropic Claude - $5 free credits monthly
│   └── Local Ollama - unlimited but slower fallback
│
└── Smart Routing Strategy
    ├── Simple commands → Hugging Face GPT-OSS-20b
    ├── Complex reasoning → Hugging Face GPT-OSS-120b
    ├── Emergency fallback → Groq/Gemini free tiers
    └── Local processing → Ollama when all APIs exhausted
```

💰 FREE TIER OPTIMIZATION STRATEGY

API Usage Optimization

```
API_OPTIMIZATION_STRATEGY:
├── Hugging Face Free Tier
│   ├── Rate limits - managed by HuggingFaceClient class
│   ├── Model selection - GPT-OSS-20b for simple, GPT-OSS-120b for complex
│   ├── Request batching - batch multiple simple requests
│   └── Response caching - cache common responses locally
│
├── Command Classification
│   ├── Local operations - file ops, app launching (no API needed)
│   ├── Simple AI tasks - basic reasoning via GPT-OSS-20b
│   ├── Complex AI tasks - advanced reasoning via GPT-OSS-120b
│   └── Emergency fallback - Groq/Gemini when HF limits reached
```

```
│
├── Usage Tracking
│   ├── APIUsageTracker class - monitors all API consumption
│   ├── DailyLimitManager class - enforces daily usage limits
│   ├── MonthlyBudgetManager class - tracks monthly spending
│   └── UsageAnalytics class - analyzes usage patterns for optimization
│
└── Cost Control Features
    ├── RequestValidator class - validates requests before sending
    ├── ResponseCache class - caches API responses for reuse
    ├── FallbackChain class - automatic fallback to free alternatives
    └── UsageAlerts class - alerts when approaching limits
```

Memory Optimization for 2GB RAM

```
MEMORY_OPTIMIZATION_2GB:
├── Core System Allocation
│   ├── OS Buffer - 800MB reserved for operating system
│   ├── Python Runtime - 200MB for Python interpreter and core libraries
│   ├── FastMCP Framework - 150MB for MCP server framework
│   └── Available Memory - 850MB for MCP servers and operations
│
├── MCP Server Memory Usage
│   ├── Central Orchestrator - 150MB (command routing and session management)
│   ├── Browser Server - 100MB (lightweight browser automation)
```

```
│   ├── Python Server - 100MB (Docker container management)
│   ├── System Operations - 80MB (file and process operations)
│   ├── Communication Server - 120MB (WhatsApp/email automation)
│   ├── IDE Integration - 80MB (VS Code and Git integration)
│   └── Shared Memory Pool - 220MB (cached data and buffers)
│
├── Optimization Techniques
│   ├── LazyLoading class - load MCP servers only when needed
│   ├── MemoryPool class - shared memory allocation across servers
│   ├── ResourceManager class - automatic memory cleanup and recycling
│   └── LightweightContainers - minimal Docker images for services
│
└── Resource Monitoring
    ├── MemoryMonitor class - real-time memory usage tracking
    ├── ResourceAlert class - alerts when memory usage exceeds thresholds
    ├── AutoScaling class - automatically scale down unused servers
    └── PerformanceProfiler class - identifies memory bottlenecks
```

🎯 DETAILED FEATURE BREAKDOWN

Natural Language Command Processing

```
COMMAND_PROCESSING_SYSTEM:
├── Command Router (core/orchestrator/src/router.py)
│   ├── CommandParser class - parse natural language into structured commands
```

```
│   ├── IntentClassifier class - classify command type and required servers
│   ├── ParameterExtractor class - extract specific parameters from commands
│   └── ExecutionPlanner class - create step-by-step execution plan
│
├── Example Command Mappings
│   ├── "Open WhatsApp and call Kartik"
│   │   ├── Step 1: SystemOps.launch_app(app_name="whatsapp")
│   │   └── Step 2: Communication.make_call(contact="Kartik")
│   │
│   ├── "Create a Python web scraper project"
│   │   ├── Step 1: IDEIntegration.create_project(name="web_scraper", type="python")
│   │   ├── Step 2: Python.execute(code="generate_scraper_boilerplate()")
│   │   └── Step 3: IDEIntegration.open_vscode(project_path="./web_scraper")
│   │
│   └── "Schedule meeting with team for tomorrow"
│       ├── Step 1: Browser.search(query="calendar scheduling")
│       ├── Step 2: Communication.send_email(to="team@company.com", subject="Meeting
Tomorrow")
│       └── Step 3: SystemOps.create_file(path="meeting_reminder.txt")
│
└── AI Processing Pipeline
    ├── HuggingFaceClient.chat_completion() - process command via GPT-OSS
    ├── HarmonyClient.format_request() - format request in Harmony format
    ├── ResponseParser.extract_actions() - extract actionable steps from AI response
    └── ExecutionEngine.execute_plan() - execute steps across MCP servers
```

System Operations Capabilities

```
SYSTEM_OPERATIONS_FEATURES:
├── Application Management
│   ├── AppLauncher.launch() - start applications by name
│   ├── ProcessMonitor.list_running() - show running processes
│   ├── ProcessKiller.terminate() - stop specific processes
│   └── ApplicationRegistry.get_app_path() - find application installation paths
│
├── File System Operations
│   ├── FileOperations.create_file() - create new files with content
│   ├── FileOperations.read_file() - read file contents
│   ├── FileOperations.write_file() - write content to existing files
│   ├── FileOperations.delete_file() - delete files and directories
│   ├── DirectoryManager.list_contents() - list directory contents
│   └── PathResolver.resolve_path() - resolve relative and absolute paths
│
├── System Information
│   ├── SystemInfo.get_cpu_info() - CPU usage and specifications
│   ├── SystemInfo.get_memory_info() - RAM usage and availability
│   ├── SystemInfo.get_disk_info() - disk space and usage
│   ├── SystemInfo.get_network_info() - network interfaces and connectivity
│   └── SystemInfo.get_os_info() - operating system details
│
```

└── Hardware Interaction

├── HardwareInterface.get_connected_devices() - list USB and Bluetooth devices

├── HardwareInterface.get_audio_devices() - list audio input/output devices

├── HardwareInterface.get_display_info() - screen resolution and displays

└── HardwareInterface.get_power_status() - battery and power information

Communication Automation

COMMUNICATION_FEATURES:

├── WhatsApp Integration

│ ├── WhatsAppWebAutomation.send_message() - send text messages

│ ├── WhatsAppWebAutomation.make_call() - initiate voice calls

│ ├── WhatsAppWebAutomation.send_media() - send images and files

│ ├── ContactManager.find_contact() - search for contacts by name

│ └── MessageQueue.schedule_message() - schedule messages for later

│

├── Email Management

│ ├── EmailClient.send_email() - send emails via Gmail API

│ ├── EmailClient.read_inbox() - read incoming emails

│ ├── EmailClient.search_emails() - search emails by criteria

│ ├── EmailClient.create_draft() - create draft emails

│ └── EmailClient.manage_labels() - organize emails with labels

│

├── Phone Integration

│ ├── PhoneCallManager.make_call() - initiate phone calls (system-dependent)

```
│   ├── PhoneCallManager.answer_call() - answer incoming calls
│   ├── PhoneCallManager.end_call() - end active calls
│   └── CallHistory.get_recent_calls() - retrieve call history
│
└── Social Media Automation
    ├── SocialMediaManager.post_twitter() - post to Twitter/X
    ├── SocialMediaManager.post_linkedin() - post to LinkedIn
    ├── SocialMediaManager.post_facebook() - post to Facebook
    └── SocialMediaManager.schedule_posts() - schedule social media posts
```

Development Integration

```
IDE_INTEGRATION_FEATURES:
├── VS Code Control
│   ├── VSCodeClient.open_project() - open projects in VS Code
│   ├── VSCodeClient.create_file() - create and open new files
│   ├── VSCodeClient.open_terminal() - open integrated terminal
│   ├── VSCodeClient.install_extension() - install VS Code extensions
│   └── VSCodeClient.run_task() - execute VS Code tasks and build scripts
│
├── Project Management
│   ├── ProjectManager.create_project() - scaffold new projects with templates
│   ├── ProjectManager.init_git_repo() - initialize Git repository
│   ├── ProjectManager.create_readme() - generate project README files
│   ├── ProjectManager.setup_dependencies() - install project dependencies
```

```
│   └── ProjectManager.create_venv() - create Python virtual environments
│
├── Git Operations
│   ├── GitOperations.init_repository() - initialize new Git repositories
│   ├── GitOperations.add_files() - stage files for commit
│   ├── GitOperations.commit_changes() - commit changes with messages
│   ├── GitOperations.push_to_remote() - push commits to remote repository
│   ├── GitOperations.pull_from_remote() - pull latest changes
│   ├── GitOperations.create_branch() - create and switch branches
│   └── GitOperations.merge_branches() - merge branches and handle conflicts
│
└── Code Analysis
    ├── CodeAnalyzer.check_syntax() - validate code syntax
    ├── CodeAnalyzer.run_linter() - run code quality checks
    ├── CodeAnalyzer.format_code() - automatically format code
    ├── CodeAnalyzer.generate_docs() - generate code documentation
    └── CodeAnalyzer.run_tests() - execute project test suites
```

🚀 IMPLEMENTATION ROADMAP

Phase 1: Core Infrastructure (Week 1)

```
PHASE_1_DELIVERABLES:
├── Setup and Configuration
│   ├── setup.sh - automated system setup and dependency installation
```

```
│   ├── install_marketplace.sh - install available MCP servers from marketplace
│   ├── docker-compose.2gb.yml - memory-optimized container configuration
│   └── .env setup - configure Hugging Face API keys and settings
│
├── Central Orchestrator (Port 9000)
│   ├── main.py - FastMCP server with command routing capabilities
│   ├── huggingface_api.py - GPT-OSS API client with rate limiting
│   ├── router.py - natural language command parsing and server routing
│   └── session.py - multi-user session management and context tracking
│
├── Available MCP Servers Integration
│   ├── Browser Server (Port 8001) - integrate existing GPT-OSS browser server
│   ├── Python Server (Port 8000) - integrate existing GPT-OSS python server
│   └── Health monitoring - verify all MCP servers are running correctly
│
└── Basic Web Interface
    ├── CommandInput component - simple text input for natural language commands
    ├── ChatInterface component - display command results and AI responses
    ├── SystemStatus component - show MCP server health and availability
    └── Basic styling - minimal CSS for functional interface
```

Phase 2: Custom MCP Servers (Week 2)

```
PHASE_2_DELIVERABLES:
├── System Operations Server (Port 8002)
```

```
│   ├── main.py - FastMCP server for system operations
│   ├── app_launcher.py - cross-platform application launching
│   ├── file_ops.py - comprehensive file system operations
│   ├── process_manager.py - process monitoring and control
│   └── system_info.py - hardware and OS information gathering
│
├── Communication Server (Port 8003)
│   ├── main.py - FastMCP server for communication automation
│   ├── whatsapp_web.py - WhatsApp Web automation using Selenium
│   ├── email_client.py - Gmail API integration for email management
│   ├── phone_calls.py - system phone integration (OS-dependent)
│   └── sms_handler.py - SMS sending and receiving capabilities
│
├── IDE Integration Server (Port 8004)
│   ├── main.py - FastMCP server for development tools
│   ├── vscode_api.py - VS Code control and project management
│   ├── git_ops.py - Git operations wrapper and automation
│   ├── project_manager.py - project scaffolding and setup
│   └── code_analysis.py - basic code quality and syntax checking
│
└── Integration Testing
    ├── test_full_workflow.py - test complete command execution chains
    ├── test_mcp_communication.py - verify MCP server intercommunication
    ├── test_memory_usage.py - ensure 2GB memory constraints are met
```

```
    └── test_api_limits.py - verify Hugging Face free tier compliance
```

Phase 3: Advanced Features (Week 3)

```
PHASE_3_DELIVERABLES:
├── Marketplace Integration
│   ├── install_github_server() - GitHub API operations MCP server
│   ├── install_git_server() - Advanced Git operations MCP server
│   ├── install_filesystem_server() - Extended file system operations
│   ├── install_slack_server() - Slack messaging integration
│   └── install_docker_server() - Docker container management
│
├── Advanced Command Processing
│   ├── MultiStepPlanner class - handle complex multi-step commands
│   ├── ContextManager class - maintain conversation context across commands
│   ├── ErrorHandler class - graceful error handling and recovery
│   └── CommandHistory class - track and replay previous commands
│
├── Performance Optimization
│   ├── ResponseCache class - cache API responses to reduce calls
│   ├── RequestQueue class - queue and batch API requests efficiently
│   ├── MemoryOptimizer class - optimize memory usage across MCP servers
│   └── ResourceMonitor class - monitor and alert on resource usage
│
└── Enhanced UI Features
```

```
    ├── Command suggestions - suggest commands based on context
    ├── Real-time status - live updates of command execution progress
    ├── Error visualization - clear display of errors and suggested fixes
    └── Command history - browsable history of executed commands
```

Phase 4: Production Ready (Week 4)

```
PHASE_4_DELIVERABLES:
├── Security and Reliability
│   ├── SecurityManager class - API key management and validation
│   ├── AuthenticationHandler class - user authentication and authorization
│   ├── InputSanitizer class - validate and sanitize user inputs
│   └── AuditLogger class - comprehensive activity logging
│
├── Monitoring and Analytics
│   ├── SystemMonitor class - real-time system health monitoring
│   ├── APIUsageAnalytics class - track and analyze API usage patterns
│   ├── PerformanceMetrics class - measure response times and success rates
│   └── UsageReports class - generate usage reports and recommendations
│
├── Documentation and Examples
│   ├── Complete API documentation - document all MCP server endpoints
│   ├── User guide - comprehensive usage guide with examples
│   ├── Developer documentation - guide for extending and customizing
│   └── Troubleshooting guide - common issues and solutions
```

```
│
└── Deployment Options
    ├── docker-compose.prod.yml - production deployment configuration
    ├── kubernetes manifests - Kubernetes deployment for scalability
    ├── cloud deployment scripts - automated cloud deployment
    └── backup and recovery - data backup and disaster recovery procedures
```

📊 EXPECTED PERFORMANCE METRICS

System Performance

```
PERFORMANCE_TARGETS_2GB:
├── Memory Usage
│   ├── Total System Memory - 2GB maximum
│   ├── Available for MCP Servers - 850MB allocated
│   ├── Peak Memory Usage - <1.8GB under normal load
│   └── Memory Efficiency - >85% utilization efficiency
│
├── Response Times
│   ├── Simple Commands - <3 seconds (file operations, app launching)
│   ├── AI-Powered Commands - <8 seconds (via Hugging Face GPT-OSS)
│   ├── Complex Multi-Step - <15 seconds (multiple MCP server coordination)
│   └── Emergency Fallback - <20 seconds (when using backup APIs)
│
├── API Usage Optimization
```

```
│   ├── Hugging Face Primary - 80% of AI requests
│   ├── Groq Fallback - 15% of requests (when HF limits reached)
│   ├── Gemini Emergency - 5% of requests (final fallback)
│   └── Local Ollama - Available but slower for unlimited use
│
└── Success Rates
    ├── Command Recognition - >90% accuracy for natural language parsing
    ├── Task Completion - >95% success rate for simple operations
    ├── Multi-Step Workflows - >85% success rate for complex automation
    └── Error Recovery - >80% automatic recovery from API failures
```

Free Tier Usage Tracking

```
FREE_TIER_MONITORING:
├── Hugging Face Inference API
│   ├── Rate Limits - managed by HuggingFaceClient with automatic backoff
│   ├── Usage Tracking - APIUsageTracker monitors daily/monthly consumption
│   ├── Model Selection - intelligent routing between GPT-OSS-20b and GPT-OSS-120b
│   └── Cost Prediction - estimate monthly costs based on usage patterns
│
├── Backup API Management
│   ├── Groq Free Tier - 14,400 tokens/day automatic tracking
│   ├── Gemini Free Tier - 1,500 requests/day with intelligent queuing
│   ├── Claude Free Credits - $5 monthly budget with cost tracking
│   └── Usage Alerts - notifications when approaching limits
```

```
│
├── Optimization Strategies
│   ├── Response Caching - cache common responses to reduce API calls
│   ├── Request Batching - combine multiple simple requests when possible
│   ├── Smart Routing - route simple tasks to faster, cheaper models
│   └── Fallback Chain - automatic fallback through available APIs
│
└── Cost Control Features
    ├── Daily Budgets - set and enforce daily spending limits
    ├── Usage Analytics - analyze patterns to optimize API usage
    ├── Alert System - warnings when approaching free tier limits
    └── Emergency Stop - halt API usage if limits are exceeded
```

🎯 EXAMPLE USER WORKFLOWS

Basic Command Examples

```
BASIC_COMMAND_WORKFLOWS:
├── File Management
│   ├── "Create a new file called notes.txt"
│   │   └── SystemOps.create_file(path="notes.txt", content="")
│   ├── "List all files in the current directory"
│   │   └── SystemOps.list_directory(path=".")
│   └── "Delete the old_file.txt"
│       └── SystemOps.delete_file(path="old_file.txt")
```

```
│
├── Application Control
│   ├── "Open calculator"
│   │   └── SystemOps.launch_app(app_name="calculator")
│   ├── "Start VS Code"
│   │   └── IDEIntegration.open_vscode()
│   └── "Launch WhatsApp"
│       └── SystemOps.launch_app(app_name="whatsapp")
│
├── System Information
│   ├── "Show me system information"
│   │   └── SystemOps.get_system_info()
│   ├── "List running processes"
│   │   └── SystemOps.list_processes()
│   └── "Check available memory"
│       └── SystemOps.get_memory_info()
│
└── Communication Tasks
    ├── "Send email to john@example.com with subject 'Meeting Tomorrow'"
    │   └── Communication.send_email(to="john@example.com", subject="Meeting Tomorrow")
    ├── "Send WhatsApp message to Sarah saying 'Running late'"
    │   └── Communication.send_whatsapp(contact="Sarah", message="Running late")
    └── "Call Mom"
        └── Communication.make_call(contact="Mom")
```

Advanced Automation Examples

ADVANCED_AUTOMATION_WORKFLOWS:

├── Development Project Setup

│ ├── "Create a Python web scraper project with Git"

│ │ ├── Step 1: IDEIntegration.create_project(name="web_scraper", type="python")

│ │ ├── Step 2: IDEIntegration.init_git_repo(path="./web_scraper")

│ │ ├── Step 3: Python.execute(code="create_requirements_file()")

│ │ ├── Step 4: IDEIntegration.open_vscode(project_path="./web_scraper")

│ │ └── Step 5: SystemOps.create_file(path="./web_scraper/README.md")

│ │

│ └── "Set up React project and install dependencies"

│   ├── Step 1: IDEIntegration.create_project(name="react_app", type="react")

│   ├── Step 2: Python.execute(code="npm_install_dependencies()")

│   ├── Step 3: IDEIntegration.open_vscode(project_path="./react_app")

│   └── Step 4: IDEIntegration.run_dev_server()

│

├── Communication Workflows

│ ├── "Schedule team meeting for tomorrow and send invites"

│ │ ├── Step 1: Browser.search(query="team calendar availability")

│ │ ├── Step 2: Communication.send_email(to="team@company.com", subject="Team Meeting Tomorrow")

│ │ ├── Step 3: SystemOps.create_file(path="meeting_agenda.txt")

│ │ └── Step 4: Communication.send_whatsapp(contact="Team Group", message="Meeting scheduled for tomorrow")

```
│   │
│   └── "Send project update to all stakeholders"
│       ├── Step 1: SystemOps.read_file(path="project_status.txt")
│       ├── Step 2: Communication.send_email(to="stakeholders@company.com",
subject="Project Update")
│       ├── Step 3: Communication.post_slack(channel="project-updates", message="Status
update sent")
│       └── Step 4: SystemOps.create_file(path="update_log.txt")
│
├── Research and Analysis
│   ├── "Research competitor analysis and create report"
│   │   ├── Step 1: Browser.search(query="competitor analysis 2024")
│   │   ├── Step 2: Browser.open(url="relevant_article_1")
│   │   ├── Step 3: Python.execute(code="analyze_competitor_data()")
│   │   ├── Step 4: SystemOps.create_file(path="competitor_report.md")
│   │   └── Step 5: IDEIntegration.open_vscode(project_path="competitor_report.md")
│   │
│   └── "Analyze codebase and suggest improvements"
│       ├── Step 1: IDEIntegration.analyze_project(path="./current_project")
│       ├── Step 2: Python.execute(code="run_code_quality_analysis()")
│       ├── Step 3: SystemOps.create_file(path="improvement_suggestions.txt")
│       └── Step 4: IDEIntegration.create_branch(name="improvements")
│
└── System Maintenance
    ├── "Backup important files and clean system"
```

```
│   ├── Step 1: SystemOps.create_directory(path="./backups")
│   ├── Step 2: SystemOps.copy_files(source="./important", destination="./backups")
│   ├── Step 3: SystemOps.clean_temp_files()
│   └── Step 4: SystemOps.get_disk_usage()
│
└── "Update all Git repositories and check status"
    ├── Step 1: IDEIntegration.find_git_repos(path="./projects")
    ├── Step 2: IDEIntegration.pull_all_repos()
    ├── Step 3: IDEIntegration.check_repo_status()
    └── Step 4: SystemOps.create_file(path="repo_status_report.txt")
```

🛠️ QUICK START GUIDE

One-Command Setup

```
# Clone and setup the complete AI OS
git clone https://github.com/your-repo/ai-os-monorepo.git
cd ai-os-monorepo
chmod +x deployment/scripts/setup.sh
./deployment/scripts/setup.sh --2gb-optimized
```

What the Setup Script Does

```
SETUP_SCRIPT_ACTIONS:
├── Environment Setup
│   ├── install_python_dependencies() - install Python 3.12+ and required packages
```

```
│   ├── install_node_dependencies() - install Node.js and npm packages
│   ├── install_docker() - install Docker for containerized MCP servers
│   └── setup_environment_variables() - configure API keys and settings
│
├── MCP Server Installation
│   ├── install_available_servers() - setup GPT-OSS browser and python servers
│   ├── install_marketplace_servers() - auto-install GitHub, Git, Slack servers
│   ├── build_custom_servers() - compile system ops, communication, IDE servers
│   └── configure_server_networking() - setup port forwarding and connections
│
├── Configuration
│   ├── setup_huggingface_api() - configure GPT-OSS model access
│   ├── setup_free_tier_limits() - configure API usage limits and tracking
│   ├── create_user_config() - create personalized configuration
│   └── setup_security() - configure security settings and permissions
│
└── Testing and Verification
    ├── health_check_all_servers() - verify all MCP servers are running
    ├── test_basic_commands() - run basic command tests
    ├── verify_api_connections() - test Hugging Face and backup API connections
    └── generate_setup_report() - create setup completion report
```

Manual Start for Development

Start 2GB optimized development environment

```
./deployment/scripts/start_2gb.sh


# The script starts:
# - Central Orchestrator (Port 9000)
# - Browser Server (Port 8001)
# - Python Server (Port 8000)
# - System Operations Server (Port 8002)
# - Communication Server (Port 8003)
# - IDE Integration Server (Port 8004)
# - Web Interface (Port 3000)
```

First Commands to Test

```
TEST_COMMANDS:
├── Basic Operations
│   ├── "Create a file called test.txt"
│   ├── "List files in current directory"
│   ├── "Open calculator"
│   └── "Show system information"
│
├── AI-Powered Tasks
│   ├── "Explain quantum computing in simple terms"
│   ├── "Write a Python function to reverse a string"
│   ├── "Create a TODO list for learning AI"
│   └── "Generate a professional email template"
```

```
│
├── Integration Tests
│   ├── "Create a Python project and open in VS Code"
│   ├── "Search for React tutorials and summarize"
│   ├── "Send test WhatsApp message to myself"
│   └── "Initialize Git repo and make first commit"
│
└── Complex Workflows
    ├── "Research Python web frameworks and create comparison"
    ├── "Set up development environment for machine learning"
    ├── "Analyze my recent code changes and suggest improvements"
    └── "Create project documentation and upload to GitHub"
```

📈 SUCCESS METRICS & VALIDATION

Technical Validation

```
TECHNICAL_SUCCESS_METRICS:
├── Performance Benchmarks
│   ├── Memory usage stays under 2GB during normal operation
│   ├── Command response time averages under 5 seconds
│   ├── MCP server uptime exceeds 99% during testing
│   └── API success rate exceeds 95% for all requests
│
├── Feature Completeness
```

```
│   ├── All 6 MCP servers operational and responding
│   ├── 50+ natural language commands working correctly
│   ├── Multi-step workflows executing successfully
│   └── Error handling and recovery functioning properly
│
├── Integration Testing
│   ├── Hugging Face GPT-OSS API integration working
│   ├── All backup APIs (Groq, Gemini) functional
│   ├── Web interface responsive and functional
│   └── CLI interface accepting and processing commands
│
└── Security and Reliability
    ├── API keys properly secured and managed
    ├── Input validation preventing malicious commands
    ├── Docker containers properly sandboxed
    └── Audit logging capturing all activities
```

User Experience Validation

```
USER_EXPERIENCE_METRICS:
├── Usability Testing
│   ├── New users can complete setup in under 15 minutes
│   ├── Basic commands work on first attempt 90% of the time
│   ├── Error messages are clear and actionable
│   └── Help documentation is comprehensive and accessible
```

```
│
├── Feature Adoption
│   ├── File operations used in 100% of test sessions
│   ├── Application launching used in 80% of sessions
│   ├── AI-powered commands used in 70% of sessions
│   └── Development tools used in 60% of sessions
│
├── Performance Satisfaction
│   ├── Users rate response speed as acceptable (>4/5)
│   ├── Command accuracy meets user expectations (>4/5)
│   ├── Overall system reliability rated highly (>4/5)
│   └── Learning curve considered reasonable (>3/5)
│
└── Value Proposition
    ├── Users report time savings from automation
    ├── Development workflow improvements documented
    ├── Communication efficiency gains measured
    └── Overall productivity increase reported
```

🎯 CONCLUSION

This 2GB + FREE Tier AI Operating System implementation provides a complete, production-ready solution that:

- **Runs on minimal hardware** (2GB RAM, any dual-core CPU)
- **Costs $0/month** using free API tiers from Hugging Face and backup providers

- **Provides enterprise-grade functionality** through MCP server architecture
- **Offers natural language control** over system operations, development tools, and communication
- **Scales efficiently** from personal use to small team deployments
- **Maintains security** through proper sandboxing and API key management

The modular MCP architecture ensures easy customization, while the Hugging Face GPT-OSS integration provides powerful AI capabilities without local model requirements. This makes it perfect for students, developers, small teams, and anyone wanting to explore AI operating system capabilities without significant resource investment.

Ready to revolutionize your computing experience with AI-powered automation! 🚀