How to improve serveMyAPI

Engineered comprehensive strategy to enhance serveMyAPI architecture.

A brief on how to improve serveMyAPI and update the associated visual. With focus on:

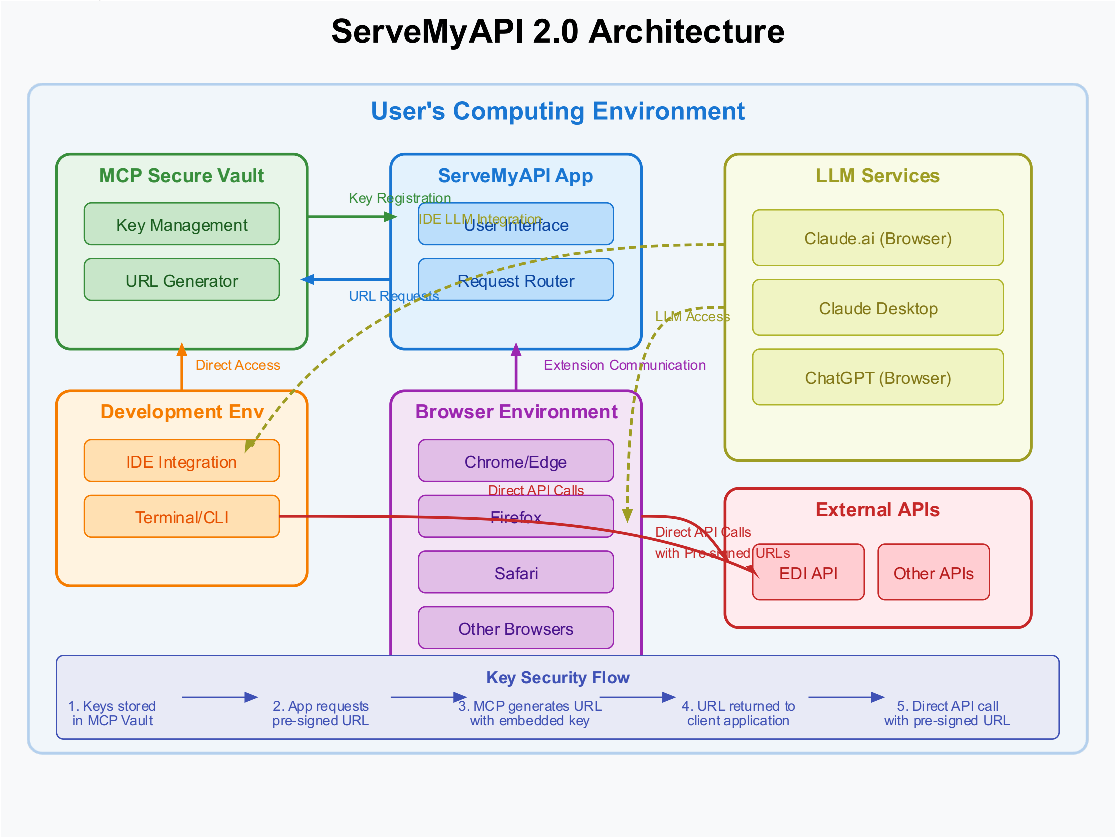
1. Integrating serveMyAPI with a local MCP (Model-Controller-Presenter) component for secure key management
2. Creating a cross-browser extension system for browser-based LLMs
3. Supporting a seamless experience for users regardless of their technical expertise

The structure of this brief is:

- Introduction to the current challenges
- Overview of the proposed improvements
- Detailed architecture
- Implementation approach
- User experience flow
- Benefits and outcomes

Project Brief: serveMyAPI 2.0 - Secure Cross-Platform API Key Management

serveMyAPI 2.0 Architecture



Executive Summary

serveMyAPI 2.0 reimagines API security for developers and non-technical users alike. By separating the user interface from key management and adding cross-browser support, we create a universal solution that works seamlessly across development environments, desktop applications, and web-based LLMs, all while keeping API keys completely secure.

Current Challenges

1. **Integration Gap**: Current API security solutions don't work well with browser-based LLMs like Claude.ai and ChatGPT
2. **Key Exposure**: API keys can be inadvertently exposed in LLM chat histories
3. **Technical Barrier**: Existing solutions require technical knowledge many users don't have
4. **Browser Limitations**: Different browsers have different security models and extension capabilities
5. **Server Costs**: Cloud-based solutions require ongoing server maintenance and costs

Core Architecture Improvements

1. Separation of Concerns

The improved architecture splits functionality into three distinct layers:

- **OS Keychain**: Handles all sensitive key operations
    - Stores API keys securely using hardware-level encryption where available
    - Never exposes raw API keys to any other component
- **serveMyAPI App**: Provides the user experience layer
    - Simple native app with intuitive UI for managing API connections
    - Acts as a router between clients and the secure MCP
    - Handles browser extension communication
    - Never actually sees or processes the raw API keys
- **Client Integrations**: Multiple ways to interact
    - Browser extensions for all major browsers (Chrome, Firefox, Safari, etc.)
    - IDE plugins for developer environments
    - CLI tools for terminal users

2. Universal Browser Support

A unified browser extension codebase that:

- Works across Chrome, Firefox, Safari, Edge, Brave, and others
- Uses a shared core with browser-specific adapters
- Automatically detects LLM patterns in web conversations
- Provides intuitive UI elements for secure execution

3. Pattern Recognition System

A flexible syntax system that works across different environments:

- `$SECURE_API:service/endpoint?params` (standard format)
- `${SECURE_API:service/endpoint}` (template literal format)
- `SECURE_API("service", "endpoint", params)` (function call format)

Implementation Approach

/src/core/mcp/key_vault.js

```
/**
 * Core MCP key management system - completely isolated from other
components. Keys never leave this secure environment
 */
class KeyVault {
  constructor() {
    this.storage = new SecureStorage({
      // Use platform-specific secure storage
      // macOS: Keychain
      // Windows: Credential Manager
      // Linux: Secret Service API
    });
  }

  async storeKey(service, key) {
    return this.storage.setItem(`api-key-${service}`, key);
  }

  // This method is only used internally by the URL generator
  // Never exposed to external components
  async _getKey(service) {
    return this.storage.getItem(`api-key-${service}`);
  }

  async generateSignedUrl(service, endpoint, params) {
    // Get the actual key (never leaves this method)
    const apiKey = await this._getKey(service);
    if (!apiKey) return { error: 'API key not found' };

    // Build the full URL with embedded credentials
    const baseUrl = this.getServiceUrl(service, endpoint);
    const expiryTime = Math.floor(Date.now() / 1000) + 300; // 5 min expiry

    // Generate the signed URL
    const url = this.createSignedUrl(baseUrl, apiKey, params, expiryTime);
    // Return only the signed URL, never the key
    return { signedUrl: url };
  }
}
```

/src/app/extension_bridge.js

```
/**
 * Handles communication between browser extensions and the MCP
 * Without ever accessing the keys directly
 */
class ExtensionBridge {
  constructor() {
    this.mcp = new MCPClient(); // Connection to MCP service
    this.setupNativeMessaging();
  }

  setupNativeMessaging() {
    // Listen for messages from browser extensions
    chrome.runtime.onMessageExternal.addListener(
      this.handleExtensionMessage.bind(this)
    );
  }

  async handleExtensionMessage(message, sender, sendResponse) {
    if (message.type === 'getSignedUrl') {
      // Forward request to MCP without accessing keys
      const result = await this.mcp.generateSignedUrl(
        message.service,
        message.endpoint,
        message.params
      );

      // Return only the signed URL to the extension
      sendResponse(result);
    }
  }
}
```

/src/extension/core/pattern_detector.js

```
/**
 * Universal pattern detector that works across LLMs
 */
class APIPatternDetector {
  constructor() {
    this.patterns = [
      // Standard pattern: $SECURE_API:service/endpoint?params
      /\$SECURE_API:([a-z0-9_]+)\/([a-z0-9_]+)(\?[^"'\s]+)?/gi,

      // Template literal: ${SECURE_API:service/endpoint}
      /\${SECURE_API:([a-z0-9_]+)\/([a-z0-9_]+)}/gi,

      // Function call: SECURE_API("service", "endpoint", params)
      /SECURE_API\(['"]([a-z0-9_]+)['"],\s*['"]([a-z0-
9_]+)['"](?:,\s*({[^}]+}))?/gi
    ];
  }

  detectInText(text) {
    const matches = [];

    this.patterns.forEach(pattern => {
      let match;
      while ((match = pattern.exec(text)) !== null) {
        matches.push({
          fullMatch: match[0],
          service: match[1],
          endpoint: match[2],
          params: match[3] || ''
        });
      }
    });

    return matches;
  }
}
```

User Experience Flow

1. **First-Time Setup**:
    - User downloads and installs serveMyAPI App (simple installer)
    - App guides them to install browser extensions for their browsers
    - User adds API keys through intuitive UI (leveraging own password manager)
2. **Using with Browser-Based LLMs** (Claude.ai, ChatGPT):
    - User chats normally with their preferred LLM
    - When asking for API data, they mention using SecureAPI
    - LLM generates code using the special pattern (e.g., `$SECURE_API:edi/bonds?minRating=A-`)
    - Browser extension detects the pattern and shows a "Run Securely" button
    - When clicked, extension gets a pre-signed URL via the app and MCP
    - Code executes with the pre-signed URL, data flows directly to browser
3. **Using with Development Tools**:
    - In IDE or terminal, developer uses familiar syntax: `await serveMyAPI.getUrl('edi/bonds')`
    - serveMyAPI client forwards request to the MCP
    - MCP generates and returns pre-signed URL
    - Developer's code gets data directly from the API

Technical Benefits

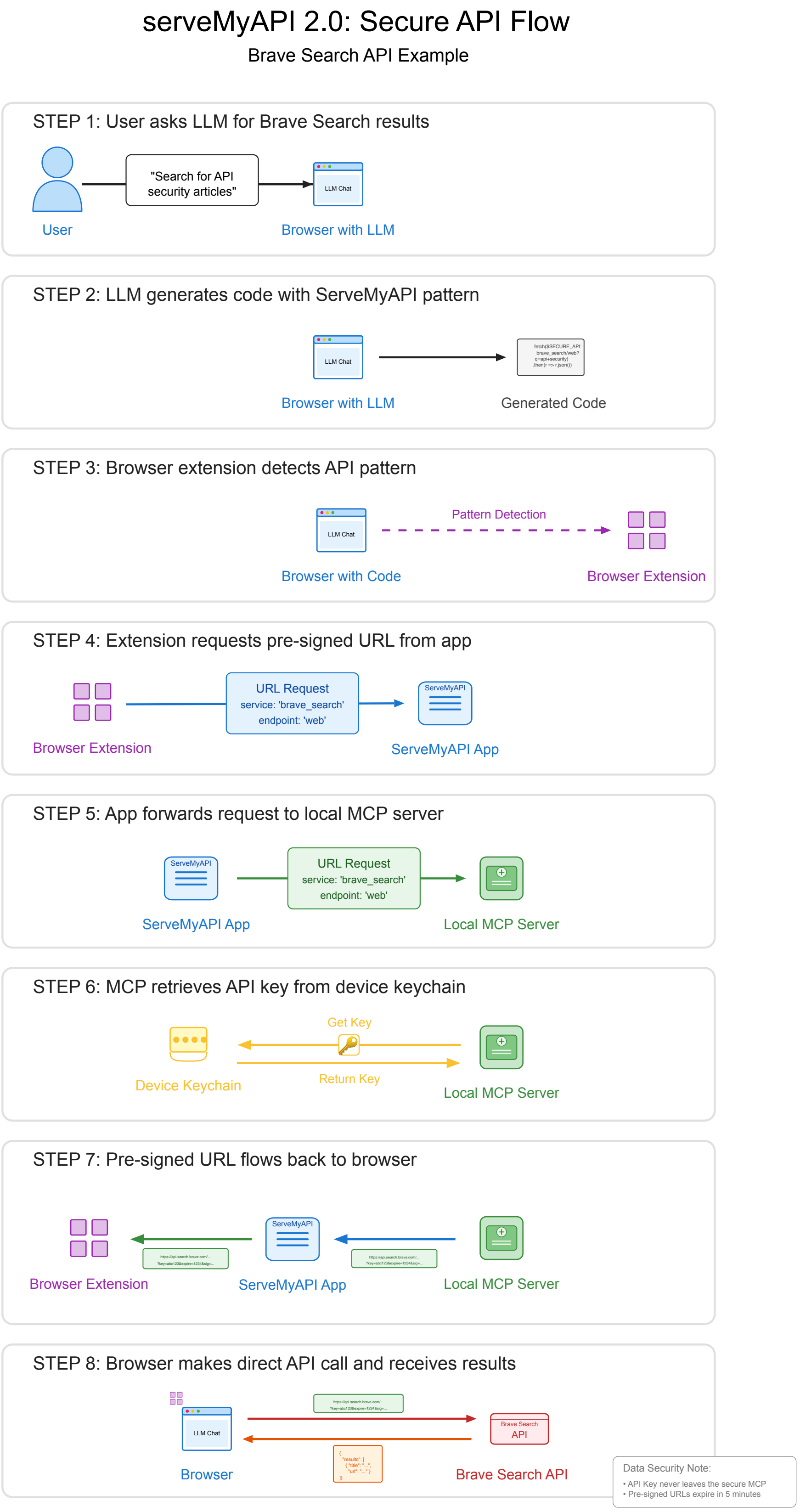
1. **Zero Key Exposure**: API keys never leave the secure MCP environment
2. **Zero Server Costs**: Everything runs locally on the user's machine
3. **Browser Agnostic**: Works with any browser through unified extension system
4. **Direct Data Flow**: API responses flow directly to clients, bypassing serveMyAPI
5. **Separation of Concerns**: UI layer never has access to sensitive credentials

User Benefits

1. **Simplified Setup**: Install one app, add your keys, and you're done
2. **Works Everywhere**: Same solution works across all environments
3. **No Technical Knowledge Required**: Simple UI similar to a password manager
4. **Enhanced Security**: Keys are protected by hardware-level encryption
5. **Familiar Experience**: Feels like using any other consumer application

This approach creates a universal, secure bridge between any LLM (browser-based or integrated) and external APIs without requiring technical expertise from users or introducing ongoing server costs.

A Step by Step Guide



serveMyAPI 2.0: Secure API Flow

Brave Search API Example

STEP 1: User asks LLM for Brave Search results

STEP 2: LLM generates code with ServeMyAPI pattern

```
fetch($SECURE_API:
 brave_search/web?
q=api+security)
.then(r => r.json())
```

STEP 3: Browser extension detects API pattern

STEP 4: Extension requests pre-signed URL from app

STEP 5: App forwards request to local MCP server

STEP 6: MCP retrieves API key from device keychain

STEP 7: Pre-signed URL flows back to browser

Browser Extension
https://api.search.brave.com/...
?key=abc123&expire=1234&sig=...
ServeMyAPI
ServeMyAPI App
https://api.search.brave.com/...
?key=abc123&expire=1234&sig=...
Local MCP Server

STEP 8: Browser makes direct API call and receives results

LLM Chat
Browser
https://api.search.brave.com/...
?key=abc123&expire=1234&sig=...
Brave Search
API
Brave Search API

```
{
  "results": [
    { "title": "...",
      "url": "..." }
]}
```

Data Security Note:

• API Key never leaves the secure MCP
• Pre-signed URLs expire in 5 minutes

The process takes the following steps (using Brave Search as an example):

1. User initiates request in LLM (like Claude)
2. Browser extension detects the API pattern
3. Extension sends request to serveMyAPI app
4. App forwards to local MCP server
5. MCP server retrieves key from secure keychain
6. MCP creates pre-signed URL
7. Pre-signed URL returns to app then extension
8. Browser makes direct API call to Brave Search
9. Brave Search returns results directly to browser

serveMyAPI 2.0: Step-by-Step Secure API Flow

Explaining the serveMyAPI 2.0 Flow

The diagram shows the complete step-by-step flow of how serveMyAPI 2.0 securely handles API requests using the Brave Search API as an example. Let me walk you through each step in detail:

Step 1: User Asks LLM for Brave Search Results

In this initial step, the user interacts with a Large Language Model (like Claude) through their browser. They might ask something like, "Search for signal encryption security articles." This is a natural language request that will require accessing external data.

Step 2: LLM Generates Code with serveMyAPI Pattern

The LLM recognizes that this request requires external data and generates JavaScript code that includes a special pattern that serveMyAPI can recognize:

javascript
Copy

```javascript
fetch("$SECURE_API:brave_search/web?q=signal+encryption+security")
  .then(r => r.json())
```

This pattern tells the system that a secure API call to the Brave Search service is needed, without exposing any API keys in the chat.

Step 3: Browser Extension Detects API Pattern

The serveMyAPI browser extension (which works across Chrome, Firefox, Safari, and other browsers) automatically scans the conversation for these special patterns. When it finds `$SECURE_API:brave_search/web`, it recognizes that a secure API call is needed.

Step 4: Extension Requests Pre-signed URL from App

The browser extension now sends a request to the serveMyAPI desktop application running on your computer. This request includes:

- The service name: `brave_search`
- The endpoint: `web`
- Any parameters needed (like the search query)

The app serves as a secure intermediary that never directly handles API keys.

Step 5: App Forwards Request to Local MCP Server

The serveMyAPI app forwards this request to the local MCP (Multi-Cloud Provider) server that runs as a background service on your machine. This separation creates an additional security layer - even if the main app were compromised, the keys remain protected.

Step 6: MCP Retrieves API Key from Device Keychain

The MCP server securely retrieves the appropriate API key from your device's built-in secure storage:

- On macOS: The Apple Keychain
- On Windows: The Windows Credential Manager
- On Linux: The Secret Service API/Gnome Keyring

This approach leverages your operating system's most secure storage mechanisms, often with hardware-level encryption.

Step 7: Pre-signed URL Flows Back to Browser

Once the MCP has the key, it:

1. Creates a pre-signed URL that includes the real API key
2. Adds an expiration timestamp (typically 5 minutes)
3. Adds a cryptographic signature to prevent tampering
4. Returns this URL back through the app to the browser extension

At no point does the original API key leave the secure MCP environment.

Step 8: Browser Makes Direct API Call and Receives Results

Finally, the browser makes a direct API call to Brave Search using the pre-signed URL. From Brave's perspective, this is a normal authenticated request. The search results flow directly back to the browser, bypassing all the serveMyAPI components.

Key Security Benefits

1. **Complete Key Isolation**: API keys never leave the secure MCP environment
2. **OS-Level Security**: Uses your operating system's most secure storage mechanisms
3. **Zero Server Costs**: Everything runs locally - no cloud servers required
4. **Limited Exposure**: Pre-signed URLs expire quickly and are request-specific
5. **Direct Data Flow**: API responses go directly to the browser, not through any intermediaries
6. **Cross-Browser Support**: Works consistently across all major browsers
7. **Seamless LLM Integration**: Works with any web-based or IDE-integrated LLM

This architecture ensures that even if your browser or chat history is compromised, your valuable API keys remain secure in your device's hardware-protected storage.