Perplexity VSCode Extension - Technical Design Document

Project Overview
- **Project Name:** Perplexity Pro Extension for VSCode
- **Version:** 1.0.0
- **Target:** Professional developers seeking agent-compatible AI research tools
- **Timeline:** 12-16 weeks (MVP to Production)
- **Team Size:** 2-3 developers

Executive Summary
This document outlines the technical architecture for a professional-grade VSCode extension that integrates Perplexity AI with advanced agent capabilities, MCP server support, and deep workspace integration. The extension addresses current market gaps by providing a robust, scalable solution for AI-assisted development workflows.

Architecture Overview

High-Level Architecture
Die Architektur ist in logische Schichten innerhalb des VSCode Extension Hosts unterteilt:

* **VSCode Extension Host**
    * **Extension Controller (`extension.ts`)**
        * Command Palette Integration
        * Status Bar Provider
        * Configuration Manager
    * **Core Services Layer**
        * PerplexityClient (API Wrapper)
        * MCPServer (Agent Mode Integration)
        * ContextManager (Workspace Analysis)
        * SecurityManager (API Key & Validation)
    * **UI Components Layer (React Webview)**
        * ChatInterface (Primary UI)
        * ResultsRenderer (Rich Content Display)
        * SettingsPanel (Configuration UI)
        * ToolsPanel (Agent Tools Interface)
    * **Tools & Providers Layer**
        * SearchProvider (Perplexity Search)
        * CodeAnalysisProvider (Workspace Integration)
        * DocumentationProvider (API Docs Integration)
        * DiagramProvider (Visual Generation)

Technology Stack

* **Backend (Extension Host):**
    * TypeScript 5.2+
    * VSCode Extension API 1.85+
    * Node.js 18+ (for MCP Server)
    * `@modelcontextprotocol/sdk` for MCP implementation
* **Frontend (Webview):**
    * React 18+ with TypeScript
    * Tailwind CSS 3.4+ for styling
    * `@vscode/webview-ui-toolkit` for native VSCode components
    * Monaco Editor integration for code display

```
* **APIs & External Services:**
    * Perplexity Sonar API (sonar-pro, sonar-medium-online)
    * Model Context Protocol (MCP) 1.0
    * VSCode Language Model Tools API
    * GitHub API (for repository context)

---

## Core Components

### 1. Extension Controller (`extension.ts`)
- **Responsibilities:** Extension lifecycle management, command
registration, VSCode API integration, global state management.
- **Key Methods:**
```typescript
export function activate(context: ExtensionContext): void;
export function deactivate(): void;

class ExtensionController {
    private chatProvider: ChatProvider;
    private mcpServer: MCPServer;
    private contextManager: ContextManager;

    async initialize(): Promise<void>;
    async handleCommand(command: string, ...args: any[]): Promise<void>;
    async dispose(): Promise<void>;
}
```

2. Perplexity API Client

- **Responsibilities:** Secure API communication, request/response handling, rate limiting, multi-model support.
- **Interface:**

TypeScript

```
interface PerplexityClient {
    search(query: string, options?: SearchOptions): Promise<SearchResult>;
    chat(messages: ChatMessage[], model?: string): Promise<ChatResponse>;
    generateSummary(content: string): Promise<SummaryResult>;
    validateApiKey(key: string): Promise<boolean>;
}

interface SearchOptions {
    model?: 'sonar-pro' | 'sonar-medium-online' | 'sonar-medium-chat';
    maxTokens?: number;
    temperature?: number;
    contextWindow?: number;
    searchDomains?: string[];
    excludeDomains?: string[];
    recency?: 'day' | 'week' | 'month' | 'year';
}
```

3. MCP Server Implementation

- **Responsibilities:** Model Context Protocol server implementation, tool registration and execution, agent mode compatibility.
- **Core Tools:**
```

TypeScript

```
interface MCPTool {
    name: string;
    description: string;
    inputSchema: JSONSchema;
    execute(args: any): Promise<ToolResult>;
}

// Primary Tools
class PerplexitySearchTool implements MCPTool {}
class WorkspaceAnalysisTool implements MCPTool {}
class CodeExplanationTool implements MCPTool {}
class DocumentationTool implements MCPTool {}
class DiagramGenerationTool implements MCPTool {}
```

4. Context Manager

- **Responsibilities:** Workspace analysis, active file context extraction, Git info, project structure understanding.
- **Key Features:**

TypeScript

```
interface WorkspaceContext {
    projectType: string;
    languages: string[];
    frameworks: string[];
    activeFiles: FileContext[];
    gitInfo: GitContext;
    dependencies: DependencyInfo[];
    codeSymbols: SymbolInfo[];
}

class ContextManager {
    async analyzeWorkspace(): Promise<WorkspaceContext>;
    async getActiveFileContext(): Promise<FileContext>;
    async getRelevantCode(query: string): Promise<CodeSnippet[]>;
    async updateContext(): Promise<void>;
}
```

5. React Webview Interface Components

- **Architecture:**

TypeScript

```
// Main Chat Interface
interface ChatInterfaceProps {
    onMessage: (message: string) => void;
    messages: ChatMessage[];
    isLoading: boolean;
    tools: AvailableTool[];
}

// Results Renderer with Rich Content
interface ResultsRendererProps {
    result: SearchResult;
    format: 'markdown' | 'structured' | 'code';
```

```
    allowInteraction: boolean;
}

// Settings Panel
interface SettingsPanelProps {
    config: ExtensionConfig;
    onConfigChange: (config: Partial<ExtensionConfig>) => void;
    apiKeyStatus: 'valid' | 'invalid' | 'unconfigured';
}
```

Security Architecture

API Key Management

- Encrypted storage using VSCode `SecretStorage` API.
- Key validation and rotation support.
- Environment variable fallback.

Input Validation

- All user inputs sanitized and validated.
- XSS protection in webview content.
- Path traversal protection for file operations.

Permission System

- Explicit user consent for tool execution.
- Workspace access permissions.
- Network request approvals.

Performance Optimization

Caching Strategy

TypeScript

```
interface CacheManager {
    searchCache: LRUCache<string, SearchResult>;
    contextCache: LRUCache<string, WorkspaceContext>;
    get<T>(key: string, category: CacheCategory): Promise<T | null>;
    set<T>(key: string, value: T, ttl: number): Promise<void>;
    invalidate(pattern: string): Promise<void>;
}
```

Resource Management

- Memory usage monitoring and cleanup.
- Background task throttling.
- Lazy loading of UI components.

Network Optimization

- Request deduplication.
- Connection pooling.
- Intelligent retry mechanisms.

Data Models

Core Data Structures

TypeScript

```
interface SearchResult {
    id: string;
    query: string;
    answer: string;
    sources: Source[];
    followUpQuestions: string[];
    model: string;
    timestamp: Date;
}

interface Source {
    url: string;
    title: string;
    snippet: string;
    domain: string;
    relevanceScore: number;
}

interface ChatMessage {
    id: string;
    role: 'user' | 'assistant' | 'system';
    content: string;
    timestamp: Date;
    tools?: ToolExecution[];
}

interface ToolExecution {
    tool: string;
    input: any;
    output: any;
    success: boolean;
    error?: string;
}
```

Configuration Management

Extension Configuration Schema

TypeScript

```
interface ExtensionConfig {
    // API Configuration
    apiKey: string;
```

```
    defaultModel: PerplexityModel;
    maxTokens: number;

    // UI Preferences
    theme: 'auto' | 'light' | 'dark';
    fontSize: number;

    // Feature Toggles
    enableMCP: boolean;
    enableWorkspaceAnalysis: boolean;

    // Security Settings
    allowFileAccess: boolean;
    trustedDomains: string[];

    // Performance Settings
    cacheSize: number;
    requestTimeout: number;
}
```

Testing Strategy

Unit Testing (Jest + TypeScript)

TypeScript

```
// API Client Tests
describe('PerplexityClient', () => {
    test('should handle search requests correctly');
    test('should validate API keys properly');
    test('should manage rate limiting');
    test('should handle network errors gracefully');
});

// MCP Server Tests
describe('MCPServer', () => {
    test('should register tools correctly');
    test('should execute tools securely');
    test('should handle tool failures');
});
```

Integration & E2E Testing

- **Integration:** VSCode Extension Host integration, Webview communication, API endpoints.
- **End-to-End:** User workflow scenarios, performance benchmarking, security vulnerability testing.

Error Handling & Logging

Error Classification

TypeScript

```
enum ErrorType {
```

```
    API_ERROR = 'api_error',
    VALIDATION_ERROR = 'validation_error',
    NETWORK_ERROR = 'network_error',
    PERMISSION_ERROR = 'permission_error',
    INTERNAL_ERROR = 'internal_error'
}

class ExtensionError extends Error {
    constructor(
        message: string,
        public type: ErrorType,
        public code?: string,
        public details?: any
    ) {
        super(message);
    }
}
```

Logging Strategy

- Structured logging with correlation IDs.
- Different log levels (debug, info, warn, error).
- User privacy protection.

Deployment & Distribution

Build Process

JSON

```
{
  "scripts": {
    "vscode:prepublish": "npm run compile && npm run build:webview",
    "compile": "tsc -p ./",
    "build:webview": "cd webview && npm run build",
    "test": "npm run compile && node ./out/test/runTest.js",
    "package": "vsce package",
    "publish": "vsce publish"
  }
}
```

Release Strategy

- Semantic versioning (e.g., 1.0.0 → 1.1.0).
- Staged rollout through VSCode Marketplace.
- Automated CI/CD pipeline with GitHub Actions.

Future Enhancements

- **Phase 2:** Multi-provider support (OpenAI, Claude), custom tool framework, team features.
- **Phase 3:** Enterprise SSO, custom deployments, advanced analytics dashboard.

Development Guidelines

- **Code Style:** TypeScript strict mode, ESLint + Prettier.
- **Git Workflow:** Feature branches, conventional commits, required code reviews.
- **Documentation:** Inline comments, API reference, user guides, Architecture Decision Records (ADRs).

Document Version: 1.0 | Last Updated: October 2, 2025 | Next Review: October 16, 2