Comprehensive Technical Blueprint for Sequential AI System Development

1. Executive Summary and Foundational Prerequisites

This report details the architectural planning, component selection, resource requirements, and implementation strategies for the sequential development of three advanced artificial intelligence applications: a Text-to-Video Generator, a Document Q&A System leveraging Retrieval-Augmented Generation (RAG), and an Autonomous Browser Agent. The intent of this blueprint is to transition system development from the information gathering phase into the building phase by providing exhaustive technical specifications for each project.

1.1 Project Overview and Strategic Sequencing Rationale

The strategic sequence for implementation is engineered to prioritize the mastery of foundational data orchestration and LLM integration techniques before committing significant capital to specialized hardware. The recommended development order is:

1. **Document Q&A System (RAG Application)**
2. **Autonomous Browser Agent (Web Automation with AI)**
3. **Text-to-Video Generator (AI Video Synthesis)**

This sequence minimizes initial capital risk associated with high-end graphical processing units (GPUs) and ensures that core competencies in data engineering and prompt engineering are validated. The RAG system provides a low-entry-barrier approach to leveraging Large Language Models (LLMs) without demanding the high computational power required for generating complex media.[1] Successful completion of the RAG system and the Autonomous Agent (Projects II and III) validates the necessary architectural and integration skills prior to the substantial investment in the GPU resources critical for the Text-to-Video Generator (Project I).[2]

1.2 Mandatory Environment Setup and Dependency Management

The stability and maintainability of these advanced AI systems rely heavily on proper dependency management. Given that each project utilizes specialized frameworks (PyTorch/Diffusion Models, LlamaIndex, Playwright), which often have strict or conflicting Python library requirements, the principle of isolation is mandatory.

The environment must be structured using lightweight Python virtual environments (venv).[3] Virtual environments ensure that each project possesses its own independent set of Python packages and a dedicated interpreter, preventing dependency conflicts (e.g., a specific PyTorch version required by ModelScope [4] conflicting with a requirement for LangChain).

The standard command structure for initiation is: python -m venv.venv within the project directory, followed by environment activation (e.g., source.venv/bin/activate or platform equivalent).[3] The resulting environment directory (.venv) should be explicitly excluded from source control (e.g., Git) as it is considered disposable and easily recreated from a configuration file.[3]

1.3 API Key Management and Security Best Practices

A foundational requirement for all three projects, particularly the RAG system and the Autonomous Agent, is the secure handling of sensitive credentials. Authentication keys for LLM providers (e.g., OpenAI, Google) and specialized services (e.g., browser-use cloud access) must never be hard-coded into the codebase or checked into source control systems.[5]

Credentials must be managed exclusively through environment variables, typically loaded at runtime from an isolated configuration file (e.g., .env).[6] The specific keys required for the stack include: BROWSER_USE_API_KEY, GOOGLE_API_KEY, or OPENAI_API_KEY for the Autonomous Agent [6], and similar keys for the RAG system's LLM back-end.

In a production context, adherence to security protocols dictates the monitoring of API key usage and a policy for regular key rotation.[7] Monitoring enables the immediate identification and mitigation of unexpected usage, which is critical if a key is suspected of being compromised.[7]

2. Project I: Text-to-Video Generator (AI Video Synthesis)

The Text-to-Video Generator, based on the components specified, is the most computationally demanding of the three projects. It serves as the capstone project, requiring substantial computational resources.

2.1 Architectural Deep Dive: Diffusion Models and Multimodal Generation

The generator operates on the principles of **Diffusion Models**, which are currently recognized as the state-of-the-art for generating high-fidelity images and videos from textual prompts [Image 3]. This process is inherently multimodal, requiring the system to understand complex linguistic instructions and translate them into visual temporal sequences.

- **Core Technology Stack:** The foundation of the system is built upon **Python**, leveraging the high-performance capabilities of the **PyTorch** framework for numerical computation and gradient tracking [Image 3]. **Transformers** are integral to the architecture, typically utilized in the text-encoder component to map the input text prompt into a latent space representation that guides the video synthesis process within the diffusion model's UNet architecture [Image 3].
- **Model Selection:** The specified implementation targets the Damo-VilAB model, accessible through the ModelScope library.[4] This specific choice demonstrates an engagement with complex, open-source video synthesis models, requiring the developer to manage large pre-trained weights.

2.2 Critical Hardware and Resource Planning

The primary constraint for video generation is not the complexity of the code, but the vast memory footprint required by the model weights and the high-resolution intermediate tensors generated during the iterative diffusion process. This project is critically constrained by **GPU VRAM**.

- **VRAM Analysis and Requirements:** Successful, stable generation of moderate-quality, short videos (e.g., 2-5 seconds) using models like Damo-VilAB necessitates a minimum of **16 GB of GPU RAM**.[2] This capacity is required to hold the model weights, optimize the computation graph, and manage the memory overhead of multi-frame generation.
- **Minimum Viable Tier:** While optimal performance dictates 16 GB VRAM, lower-end consumer GPUs with **8 GB VRAM** (e.g., an Intel Arc A750 or specific RTX 3060 models) can be used, but this requires significant compromises.[8] These compromises include reducing the output resolution, lowering the frame count, and mandatory utilization of low-VRAM optimization flags.[9]
- **Performance Implications:** Generating a short video on a high-end card (e.g., RTX 3090) typically takes approximately two to five minutes.[10] The use of minimum-tier hardware (8 GB) extends these generation times significantly, as the system must swap data more frequently or process smaller batches. This observation reinforces the project's position as the final phase, ensuring that the necessary hardware investment is validated by prior successful project completions.

The following table summarizes the VRAM requirements based on complexity:

GPU VRAM Tiers for Text-to-Video Synthesis

Hardware Tier	**VRAM Requirement (FP16/BF16)**	**Example GPU**	**Expected Performance (Damo-VilAB)**
Minimum Viable	8 GB	RTX 3060, Intel Arc A750	Low Resolution/Frames, Requires Optimization Flags
Optimal Developer	12 GB – 16 GB	RTX 3080, V100, A10	Standard Quality, Generation time $\approx$ 2-5 minutes
High Production	24 GB+	RTX 4090, A100	High Quality, High Resolution, Batch Processing

- **VRAM Calculation Principle:** VRAM usage is fundamentally driven by the number of model parameters and the numerical precision used. Utilizing half-precision floating-point numbers (FP16 or BF16, 2 bytes per parameter) is essential for large models.[11] For example, a model with 7 billion parameters (7×10^9) requires $14

\text{ GB}$ of VRAM just to store the weights in FP16 precision ($7 \times 10^9 \text{ parameters} \times 2 \text{ bytes/parameter}$).[11]

2.3 Implementation and Installation Guide

Implementation begins with strict adherence to the specified software versions to avoid dependency conflicts, which are common in deep learning stacks.

1. **Dependency Installation:** Within the isolated virtual environment, the required packages must be installed precisely as specified by the model maintainers: pip install modelscope==1.4.2 open_clip_torch pytorch-lightning.[2] The explicit version pinning of ModelScope (1.4.2) indicates that the model exhibits version fragility and relies on specific APIs provided by that release.
2. **Model Acquisition:** The large model weights are secured using the snapshot_download utility from the huggingface_hub library. The weights must be downloaded to a local directory (e.g., weights) that is then referenced by the synthesis pipeline.[2]
3. **Pipeline Execution:** Once the weights are local, the ModelScope pipeline is initialized, pointing to the local directory. The user then defines the text prompt as a dictionary input to the pipeline function, which returns the path to the generated video file.[4] If dedicated hardware is not immediately available, cloud GPU instances (such as those offering V100 or A10 GPUs) provide a viable path to test the pipeline before purchasing dedicated hardware.[13]

3. Project II: Document Q&A System (RAG Application)

The Document Q&A system is the first project in the development sequence, focusing on building a robust Retrieval-Augmented Generation (RAG) pipeline to interface LLMs with custom knowledge bases (PDFs, websites) [Image 1].

3.1 RAG System Architecture: LlamaIndex and LangChain Integration

The RAG architecture combines information retrieval with LLM generation, enabling context-aware responses that are grounded in specific documents.[1] This technique is highly

effective for reducing model hallucination and eliminating the need for expensive LLM fine-tuning.[1]

- **LlamaIndex as the Orchestrator:** LlamaIndex is the central framework for this project, simplifying the complex RAG workflow. It handles document parsing, indexing (creating the vector store index), and creating the query engine necessary to synthesize answers from the retrieved context.[1] LlamaIndex provides abstractions to manage unstructured documents (PDFs, HTML), as well as semi-structured data (text-to-SQL, text-to-Pandas).[14]
- **Role of LangChain:** While LlamaIndex is sufficient for the core Q&A task, LangChain can be integrated to handle higher-level, multi-step application logic and complex data orchestration, providing architectural flexibility for future enhancements.[15]

3.2 Advanced Document Preprocessing and Chunking Strategies

The performance of any RAG system is directly proportional to the quality of its retrieval component, which is critically dependent on how documents are preprocessed and segmented, known as chunking.

- **Structural Parsing for Documents:** Standard text extraction often fails with complex file types like PDFs containing mixed content (charts, tables). Utilizing advanced tools like LlamaParse is recommended, as it is specifically designed to handle the challenging extraction and structural integrity of data from such elements.[16]
- **Optimizing Retrieval Accuracy via Chunking:** The choice of chunking strategy is fundamental to balancing retrieval relevance against the context provided for synthesis.
    - **Sliding Window Chunking:** This method creates overlapping chunks by moving a fixed-size window across the text.[17] This is crucial for documents (like narrative reports or patient notes) where context continuity is required across chunk boundaries to ensure that retrieval captures complete ideas.[17]
    - **Document-Based Chunking:** This strategy uses the intrinsic document structure (e.g., Markdown headings, HTML tags, or code functions) as separators.[18] This ensures that logical units of information are kept intact, making it ideal for manuals, reports, or programming documentation.[18]

3.2.1 Decoupled Retrieval and Synthesis Context

A key architectural enhancement for production-grade RAG is the decoupling of chunks used

for retrieval from those used for synthesis.[19] This strategy resolves a fundamental trade-off:

1. **Retrieval Chunks:** These are typically small, dense embeddings optimized for semantic search accuracy. Small chunks minimize the "noise" or irrelevant filler content that can dilute the vector representation, leading to more precise matching.[19]
2. **Synthesis Chunks:** Once a relevant small chunk is retrieved, the corresponding large, contextual original passage is passed to the LLM.[19] This ensures the model has sufficient detail to formulate a complete and non-hallucinated answer, compensating for the brevity of the retrieval chunk.

3.3 Vector Store Selection and Performance Metrics

The vector store acts as the index for the document embeddings. The selection among FAISS, Chroma, and managed alternatives depends on the anticipated data volume and performance requirements [Image 1].

The analysis below compares the three primary options based on architecture and use case:

Vector Store Comparison for RAG Applications

Vector Store / Library	Primary Architecture	Best Use Case	Scalability Profile
Chroma [20]	Embedded Vector Database (Single Node)	Rapid Prototyping, Local Demos, Small/Medium Datasets	Easy integration, limited horizontal scaling
FAISS [20]	Similarity Search Library (Index Focus)	High-Performance, GPU-accelerated Search, Custom Indexing	High volume (Billions of vectors), maximum control
Pinecone (Managed) [15]	Fully Managed Cloud Service (API Access)	Production Systems, High Concurrency, Real-time Updates	Automated, Global Scale and Reliability

- **Initial Recommendation:** Chroma is the optimal choice for the prototyping phase and proof-of-concept development due to its lightweight, embedded-first architecture and ease of use in local Python environments.[20]
- **Scaling Requirements:** Should the project expand to enterprise scale, handling billions of vectors, **FAISS** is superior for performance-critical applications demanding GPU acceleration and maximum control over indexing strategies.[21] Alternatively, managed services like Pinecone provide automatic scaling, reliability, and real-time updates required for heavy-load production systems.[15]

3.4 Deployment and Presentation Strategy

The project stack specifies both Streamlit and FastAPI as potential deployment targets [Image 1]. The choice between them dictates the application's function (interactive demo vs. scalable service).

- **Streamlit for Rapid Prototyping:** Streamlit is the preferred tool for quickly creating an interactive web application suitable for demonstrations and proofs-of-concept (POCs).[22] It simplifies the front-end development, allowing developers to focus primarily on the RAG pipeline logic.[22]
- **FastAPI for Production API:** For deployment as a high-performance, scalable backend service, FastAPI is the superior choice. It supports asynchronous operations and automatically generates API documentation (OpenAPI/Swagger UI), making it ideal for integrating the RAG logic into larger existing applications or microservices.[22]
- **Hybrid Optimization:** The highest performance architecture involves using FastAPI to host the RAG model and core logic, while Streamlit acts as an interactive front-end client consuming the FastAPI service.[22] This maximizes both development velocity and application throughput.

4. Project III: Autonomous Browser Agent (Web Automation with AI)

The Autonomous Browser Agent focuses on creating an AI entity capable of performing complex, goal-oriented tasks on the web (searching, clicking, filling forms, scraping) without manual intervention [Image 2].

4.1 Foundations of LLM Agent Architecture and Tool Use

The operation of an autonomous agent is defined by a cyclical framework: the Agent/Brain receives a User Request, engages in Planning, uses Tools to execute actions, and manages Memory of past steps.[24]

- **Playwright and Automation:** Playwright provides the core capability for modern, robust browser control. It enables the agent to launch a browser, inspect the Document Object Model (DOM), and execute precise actions against the web interface.[25]
- **Tool Abstraction and Reasoning:** The LLM acts as the planner and reasoning core, interpreting the user's task and deciding which specialized tools to invoke.[26] The ability to leverage specific external tools is a key component of agent frameworks, supported by techniques such as Function Calling or architectures like MRKL.[24]

4.2 The Browser Use Abstraction Layer and Task Execution

The integration of LLMs with web automation is typically complex due to the volatility of element selectors. The browser-use framework addresses this by acting as an abstraction layer, simplifying the environment for the LLM.[27]

- **Simplified Interaction:** Instead of requiring the agent to deduce intricate XPath or CSS selectors, browser-use analyzes the webpage, extracts all interactive elements (buttons, links, inputs), and provides a structured interface that allows the LLM to command these actions directly in natural language.[27]
- **Agent Initialization:** The core components include instantiating the Agent with the natural language task, the selected llm (reasoning engine), and a browser instance (Playwright driver).[6]
- **Custom Tool Integration:** The capabilities of the agent are not limited to predefined web interactions. Developers can extend the agent’s functionality by defining custom functions decorated with @tools.action, allowing the LLM to call external APIs or local utilities as part of its task execution.[6]

4.3 LLM Selection and API Configuration for Agent Tasks

The choice of LLM directly influences the agent's task-completion accuracy, latency, and operational cost. The system supports integration with major LLM APIs, including OpenAI and Google Gemini [Image 2].

- **Cost-Efficiency Mandate:** Web automation tasks often involve sequential, high-frequency steps. Therefore, cost and latency optimization are critical design factors.
- **The Optimized Model (ChatBrowserUse()):** The system analysis indicates that the dedicated ChatBrowserUse() model is superior for this specific use case.[6] This model is fine-tuned for browser automation, achieving state-of-the-art accuracy, lower latency, and significantly lower operational cost—up to **15x cheaper** (53 tasks per dollar) than using general-purpose models like GPT-4 or Gemini.[28]
- **Flexibility and Local Models:** The framework supports standard LLM providers (OpenAI, Gemini, Anthropic) requiring the corresponding API key (OPENAI_API_KEY, GOOGLE_API_KEY) in the environment variables.[6] Furthermore, for local development and cost reduction during prototyping, the system integrates with **Ollama**, allowing the use of local models like llama3.1:8b via the ChatOllama class.[6]

Supported LLM Backends for Autonomous Browser Agent

LLM Backend	Integration Class	Performance in Browser Tasks	Required Configuration
ChatBrowserUse (Recommended)	ChatBrowserUse()	SOTA Accuracy, 15x Cheaper/Faster	BROWSER_USE_API_KEY [6]
OpenAI	ChatOpenAI()	High Accuracy (O3 recommended)	OPENAI_API_KEY [6]
Google Gemini	ChatGoogle()	Standard performance	GOOGLE_API_KEY [6]
Ollama (Local)	ChatOllama()	Free, requires local server setup	Model name (e.g., llama3.1:8b) [6]

4.4 Productionizing the Agent: Scalability and Stealth

Moving the agent from a local prototype to a production system requires solving infrastructure challenges related to scaling, concurrency, and handling modern web defenses.

- **Cloud Deployment via @sandbox:** To achieve scalability, the local asynchronous agent logic is wrapped using the @sandbox() decorator.[6] This delegates the management of headless browser infrastructure, memory persistence, and parallel execution to the Browser Use Cloud API, enabling the code to run at scale.[6]
- **Stealth and Anti-Bot Bypass:** Production-grade agents must be capable of handling common web defenses (CAPTCHAs, Cloudflare). The cloud infrastructure provides features like global proxy rotation (specified by cloud_proxy_country_code='us') and stealth fingerprinting, enabling the agent to browse like a human and bypass anti-bot systems.[6]
- **Persistent Profiles and Authentication:** High-value automation often requires the agent to be logged into a service. This authenticated state is managed by Persistent Profiles. The system allows synchronization of local cookies to the cloud infrastructure via a command-line utility, returning a profile_id.[6] This profile ID is then passed to the @sandbox() decorator (cloud_profile_id='your-profile-id') to ensure the cloud-executed browser session is already logged in, facilitating authenticated workflows.[6]

5. Comparative Analysis and Sequential Build Plan

5.1 Comparative Metrics: Cost, Complexity, and Deployment Effort

The three projects present distinct profiles regarding technical difficulty, resource cost, and the nature of the challenge.

Project	Primary Technical Challenge	Required Resources	Critical Success Factor	Primary Development Tool
Document Q&A (RAG)	Data engineering, retrieval pipeline	Minimal (CPU/Low-end GPU for embedding)	Retrieval Accuracy and Context Decoupling [19]	LlamaIndex / Chroma [Image 1]

	optimization (chunking, vector index)			
Autonomous Agent	LLM planning, external tool use, state management, stealth	API Access, Managed Cloud Infrastructure (Optional)	Cost Efficiency (Specialized LLMs) and Stealth [28]	Playwright / browser-use [Image 2]
Text-to-Video Generator	VRAM bottleneck, high computational throughput	High-End GPU (16GB+ VRAM minimum)	Hardware Provisioning and Model Optimization [2]	PyTorch / Diffusion Models [Image 3]

5.2 Recommended Sequential Build Path

The analysis confirms the strategic sequencing based on escalating resource demands and complexity:

1. **Phase 1: Project II (Document Q&A System):** This phase establishes the core Python environment and proficiency in LLM data workflows (RAG). By focusing on advanced chunking techniques and utilizing lightweight, embedded vector stores like Chroma, the developer gains immediate demonstrable success with minimal financial commitment.
2. **Phase 2: Project III (Autonomous Browser Agent):** This transition introduces the concept of LLMs as complex reasoning engines capable of operating statefully in an external environment. The primary focus shifts to integrating proprietary frameworks (like browser-use) and mastering cost optimization by selecting specialized LLMs for high-frequency tasks, proving capability in complex system integration and cloud scaling.
3. **Phase 3: Project I (Text-to-Video Generator):** This final phase requires the dedicated investment in high-VRAM GPU hardware. The acquired expertise in Python development, dependency management, and LLM architecture from the first two phases ensures that the high-cost computational capacity is utilized effectively for the demanding task of diffusion model synthesis.

5.3 Final Recommendations and Future-Proofing

Successful deployment of these projects requires planning for scale and modularity:

- **RAG Scalability:** While Chroma is effective for prototyping, scalability to large, production-level knowledge bases mandates a migration path to horizontally scalable vector databases, such as managed services like Pinecone or self-hosted, GPU-accelerated indexes like FAISS.[15]
- **Agent Modularity:** The architecture of the Autonomous Agent should be designed to accommodate emerging multi-agent paradigms. Systems are moving toward chained agent loops, such as those demonstrated by Playwright's Planner, Generator, and Healer agents.[29] Designing the current browser-use agent to be modular will facilitate future integration of specialized LLM components for enhanced planning and self-correction capabilities.
- **Hardware Acquisition for Video Synthesis:** If immediate purchase of a high-end 16GB VRAM GPU is unfeasible, utilizing cloud computing resources (e.g., leasing access to V100 or A10 cloud GPUs) provides a viable alternative for validating the ModelScope pipeline performance prior to permanent hardware acquisition.[13]

Works cited

1. Llamaindex RAG Tutorial - IBM, accessed November 24, 2025, https://www.ibm.com/think/tutorials/llamaindex-rag
2. Modelscope AI Text to Video | AI Video Generator, accessed November 24, 2025, https://modelscopeai.com/
3. venv — Creation of virtual environments — Python 3.14.0 documentation, accessed November 24, 2025, https://docs.python.org/3/library/venv.html
4. ali-vilab/modelscope-damo-text-to-video-synthesis - Hugging Face, accessed November 24, 2025, https://huggingface.co/ali-vilab/modelscope-damo-text-to-video-synthesis
5. Production best practices - OpenAI API, accessed November 24, 2025, https://platform.openai.com/docs/guides/production-best-practices
6. browser-use/browser-use: Make websites accessible for AI ... - GitHub, accessed November 24, 2025, https://github.com/browser-use/browser-use
7. Best Practices for API Key Safety | OpenAI Help Center, accessed November 24, 2025, https://help.openai.com/en/articles/5112595-best-practices-for-api-key-safety
8. PyTorch System Requirements - GeeksforGeeks, accessed November 24, 2025, https://www.geeksforgeeks.org/python/pytorch-system-requirements/
9. What will the hardware requirements be for txt2vid/img2vid models compared to txt2img? : r/StableDiffusion - Reddit, accessed November 24, 2025, https://www.reddit.com/r/StableDiffusion/comments/1guex2h/what_will_the_hardware_requirements_be_for/
10. ali-vilab/modelscope-damo-text-to-video-synthesis · For the people that run this locally, how long does it take to generate a video and what gpu are you using? -

Hugging Face, accessed November 24, 2025, https://huggingface.co/ali-vilab/modelscope-damo-text-to-video-synthesis/discussions/8
11. How To Calculate GPU VRAM Requirements for an Large-Language Model, accessed November 24, 2025, https://apxml.com/posts/how-to-calculate-vram-requirements-for-an-llm
12. Text to Video Free AI tools Prompt Generation with ModelScope - DAMO - YouTube, accessed November 24, 2025, https://www.youtube.com/watch?v=Udfgbb5pjpk
13. Accelerated Generative Diffusion Models with PyTorch 2, accessed November 24, 2025, https://pytorch.org/blog/accelerated-generative-diffusion-models/
14. Question-Answering (RAG) | LlamaIndex Python Documentation, accessed November 24, 2025, https://developers.llamaindex.ai/python/framework/use_cases/q_and_a/
15. How does LlamaIndex compare to other vector databases like Pinecone? - Milvus, accessed November 24, 2025, https://milvus.io/ai-quick-reference/how-does-llamaindex-compare-to-other-vector-databases-like-pinecone
16. Parsing PDFs with LlamaParse: a how-to guide - LlamaIndex, accessed November 24, 2025, https://www.llamaindex.ai/blog/pdf-parsing-llamaparse
17. RAG 2.0 : Advanced Chunking Strategies with Examples. | by Vishal Mysore | Oct, 2025, accessed November 24, 2025, https://medium.com/@visrow/rag-2-0-advanced-chunking-strategies-with-examples-d87d03adf6d1
18. Chunking Strategies to Improve Your RAG Performance - Weaviate, accessed November 24, 2025, https://weaviate.io/blog/chunking-strategies-for-rag
19. Building Performant RAG Applications for Production | LlamaIndex Python Documentation, accessed November 24, 2025, https://developers.llamaindex.ai/python/framework/optimizing/production_rag/
20. Chroma vs FAISS | Zilliz, accessed November 24, 2025, https://zilliz.com/comparison/chroma-vs-faiss
21. Chroma vs Faiss vs Pinecone: Detailed Comparison of Vector Databases - Designveloper, accessed November 24, 2025, https://www.designveloper.com/blog/chroma-vs-faiss-vs-pinecone/
22. Streamlit vs FastAPI - Kaggle, accessed November 24, 2025, https://www.kaggle.com/discussions/questions-and-answers/475580
23. Streamlit vs FastAPI: Choosing the Right Tool for Deploying Your Machine Learning Model | by Pelumi Ogunlusi | Medium, accessed November 24, 2025, https://medium.com/@samuelogunlusi07/streamlit-vs-fastapi-choosing-the-right-tool-for-deploying-your-machine-learning-model-1d16d427e130
24. LLM Agents - Prompt Engineering Guide, accessed November 24, 2025, https://www.promptingguide.ai/research/llm-agents
25. Integrating AI into Automation Testing: Part 2 — Setting Up Playwright MCP in VS Code (Step-by-Step Guide), accessed November 24, 2025, https://medium.com/@rajesh.yemul_42550/integrating-ai-into-automation-testin

g-part-2-setting-up-playwright-mcp-in-vs-code-148632da0e10
26. Introduction to Microsoft Agent Framework, accessed November 24, 2025, https://learn.microsoft.com/en-us/agent-framework/overview/agent-framework-overview
27. Build an AI Browser Agent With LLMs, Playwright, Browser-Use - DZone, accessed November 24, 2025, https://dzone.com/articles/build-ai-browser-agent-llms-playwright-browser-use
28. Browser Use - Enable AI to automate the web, accessed November 24, 2025, https://browser-use.com/
29. Playwright Test Agents, accessed November 24, 2025, https://playwright.dev/docs/test-agents
30. 🎭 Playwright AI Test Agents Explained Step-by-Step (Planner, Generator, Healer), accessed November 24, 2025, https://www.youtube.com/watch?v=fxkNt3QqiDA