AI in the Browser:

Building LLM-Powered Browser Agents

Complete Masterclass Session Outline

Duration: 2.5-3 hours (150-180 minutes)
Target Audience: Engineers with intermediate Python skills and basic familiarity with LLMs

---

Prerequisites

Participants should have:

- Working knowledge of Python (functions, classes, async/await)
- Basic understanding of web technologies (HTML, CSS, DOM)
- Familiarity with API concepts and REST patterns
- Exposure to LLMs (ChatGPT, Claude, or similar) preferred but not required
- Laptop with Python 3.9+ installed
- GitHub account for accessing course repository

Learning Objectives

By the end of this masterclass, participants will be able to:

1. **Design and implement** autonomous browser agents that make decisions using LLMs
2. **Integrate** Playwright with LLM APIs to create intelligent web automation workflows
3. **Apply** memory and context management techniques for agent continuity across sessions
4. **Evaluate** security risks and implement mitigation strategies for production agent deployments
5. **Compare** hosted vs. on-device LLM approaches and select appropriate architecture patterns

---

Session Flow

Segment 1: Welcome & Context Setting (0-15 min)

Activities:

- Introduction and learning objectives review
- Interactive poll: Who has built web scrapers? Who has used LLM APIs?
- Real-world motivation: why agents matter in 2025 (RPA evolution, autonomous research assistants, testing automation)
- Quick demo: Show a working agent booking a restaurant reservation

---

Segment 2: Foundations - Browser Automation & LLMs (15-35 min)

Activities:

- Browser automation landscape: Selenium vs Playwright vs Puppeteer (2025 comparison)
- Why Playwright for Python: async support, reliable waits, modern APIs
- LLM capabilities relevant to agents: reasoning, function calling, structured outputs
- Live demo: Simple Playwright script navigating a page

Key Takeaway: Modern browser automation + LLM reasoning = autonomous agents

---

Segment 3: Agent Architecture Deep Dive (35-60 min)

Activities:

- The agentic loop: Observe → Reason → Act → Learn
- Component breakdown: Browser Controller, LLM Client, Tool Registry, Memory Store
- Memory patterns: short-term (session state), working (context window), long-term (vector DB)
- Architecture diagram walkthrough with real traffic patterns
- Security checkpoint: what could go wrong?

Key Takeaway: Agent architecture follows observe-reason-act-learn cycle with security as a first-class concern

---

Segment 4: LLM Integration Patterns (60-75 min)

Activities:

- Hosted APIs: OpenAI Responses API, Anthropic Claude, Google Gemini
- Function calling and structured outputs (JSON mode)
- Prompt engineering for agents: system prompts, few-shot examples, chain-of-thought
- On-device options: WebLLM, Chrome's built-in APIs, quantized models
- Trade-offs: latency, cost, privacy, model capability

Key Takeaway: Choose hosted for capability, on-device for privacy; no silver bullet

---

Segment 5: Hands-On Lab - Build a Research Assistant Agent (90-135 min)

Problem Statement:
Build an agent that searches a knowledge base site, extracts key information, and summarizes findings.

Lab Structure:

Step 1: Environment Setup (10 min)

- Clone repository
- Install dependencies (Playwright, OpenAI SDK, etc.)
- Verify browser installation
- Test API key connection

Step 2: Implement Navigation with LLM Decisions (15 min)

- Load target page
- Extract available links and content
- Send to LLM for decision on which link to follow
- Execute navigation based on LLM response

Step 3: Add Extraction Logic (10 min)

- Implement structured data extraction
- Parse LLM outputs with Pydantic
- Validate extracted data against schema

Step 4: Integrate Memory (10 min)

- Maintain set of visited URLs
- Store extracted data in session state
- Pass context to LLM for continuity

Step 5: Error Handling & Retry Logic (10 min)

- Add try/except blocks for common failures
- Implement exponential backoff for rate limits
- Log all decisions and errors

Key Takeaway: Building agents is iterative; expect failures and build robustness

---

Segment 6: Production Considerations (135-155 min)

Activities:

- Testing strategies: unit tests, integration tests, replay-based testing
- Observability: logging, tracing, anomaly detection
- Deployment patterns: server-side agent services, hybrid (browser in cloud), edge computing
- Security hardening: SesameOp backdoor case study (Microsoft DART, Nov 2025), mitigation checklist
- Cost management: API rate limits, caching strategies, fallback models

Key Takeaway: Production requires testing, monitoring, security hardening, and cost controls

---

Segment 7: Advanced Topics & Future Directions (155-170 min)

Activities:

- Multi-agent systems: coordination, communication protocols
- RAG for agents: when to use vector stores, embedding strategies
- Web-based inference: Chrome's Gemini Nano, WebGPU compute
- Tool ecosystems: LangChain, AutoGen, Semantic Kernel comparison
- Emerging patterns: model-as-judge, self-healing agents

Key Takeaway: The field is rapidly evolving; continuous learning is essential

---

Segment 8: Wrap-Up & Assessment (170-180 min)

Activities:

- Review key concepts via interactive quiz (10 questions)
- Assignment briefing: extend the lab to handle multi-page workflows

- Q&A session
- Further resources and community links
- Feedback survey

Key Takeaway: You've built a working agent; now extend and productionize it

---

Key Concepts Reference

1. Browser Agent

An autonomous software system that navigates and interacts with web applications, making decisions based on real-time observations and LLM reasoning rather than following pre-programmed rules.

2. Agentic Loop

The cyclical process where an agent observes its environment (web page state), reasons about what action to take (via LLM), executes that action (browser automation), and optionally learns from the outcome to inform future decisions.

3. Function Calling (Tool Use)

LLM capability where the model outputs structured JSON indicating which function to invoke with what parameters, enabling agents to take concrete actions like clicking buttons or filling forms.

4. Playwright

A modern browser automation library from Microsoft that provides reliable, cross-browser automation APIs with built-in waits and debugging tools, particularly strong in async Python environments.

5. Context Window

The amount of text (measured in tokens) an LLM can process at once, typically 8k-128k tokens in 2025 models, constraining how much page content and conversation history an agent can reason over.

6. RAG (Retrieval Augmented Generation)

Pattern where agents retrieve relevant information from external knowledge bases (often vector databases) before generating responses, extending memory beyond the context window.

7. WebLLM

Framework for running quantized language models directly in the browser using WebGPU, enabling on-device inference with privacy benefits but limited to smaller models (sub-10B parameters).

8. Structured Output

LLM response constrained to a specific format (usually JSON schema), ensuring agents receive parseable data they can reliably act upon rather than freeform text.

9. Memory Store

System for persisting agent state across sessions, ranging from simple key-value stores for session data to vector databases for semantic retrieval of past interactions.

10. Allowlist Pattern

Security practice where agents can only execute pre-approved actions from a defined set, preventing arbitrary code execution or unintended system access.

Case Study: Automated Competitive Intelligence Agent

Problem Statement

Build an agent that monitors competitor websites (with permission), extracts pricing and feature information, compares against your product, and generates a weekly summary report. The agent should handle login (if required), navigate multiple product pages, extract structured data, and store findings in a queryable format.

Acceptance Criteria

1. Agent successfully authenticates to a demo competitor site
2. Extracts at least 5 data points per product (name, price, key features, availability)
3. Makes intelligent decisions about which links to follow (product pages vs irrelevant content)
4. Maintains extraction accuracy >90% on test dataset
5. Generates structured JSON output with timestamp and source URLs
6. Handles common error cases (page not found, captcha detection, network timeouts)
7. Completes full workflow in <5 minutes for 10 product pages
8. Logs all actions for audit trail

Constraints

- No actual competitor sites—use provided mock e-commerce demo site
- Rate limit: max 1 request per second
- Must respect robots.txt patterns
- API usage capped at 50k tokens per run

Success Metrics

- Code quality (modular, tested, documented)
- Data accuracy (validated against ground truth)
- Error handling robustness (graceful degradation)
- Security considerations implementation (allowlist, sanitization)

Tools & Libraries

1. Playwright (Python) - v1.41+

Why: Best-in-class browser automation with async support, reliable selectors, and excellent debugging tools. Maintained by Microsoft with strong Python bindings. Auto-wait functionality prevents flaky tests.

2. OpenAI Python SDK - v1.51+

Why: Industry standard for LLM API access. Supports function calling, structured outputs, and vision APIs. We'll use GPT-4 Turbo or GPT-4o for reasoning tasks. Well-documented with active community support.

3. LangChain - v0.3+

Why: Mature framework for agent orchestration with built-in memory, tools, and prompt templates. Large community and extensive documentation. Simplifies common agent patterns without being overly opinionated.

4. ChromaDB - v0.5+

Why: Lightweight vector database for storing and retrieving agent memories. Easy to run locally, no complex setup required. Python-native with good integration into agent workflows.

5. Pydantic - v2.9+

Why: Data validation and parsing using Python type annotations. Perfect for ensuring LLM outputs match expected schemas. Catches errors early and provides clear validation messages.

6. pytest + pytest-asyncio - v8.3+ / v0.24+

Why: Standard Python testing framework with async support for testing Playwright workflows and LLM interactions. Fixtures make test setup/teardown clean.

Assessment Plan

Quiz (10 Multiple-Choice Questions)

- **Format:** Multiple choice, single correct answer
- **Topics Covered:** Agent architecture, LLM capabilities, security, Playwright basics, memory patterns
- **Passing Score:** 70% (7/10 correct)
- **Timing:** In-class, 10 minutes
- **Purpose:** Reinforce key concepts, identify areas needing review

Practical Assignment

Title: Multi-Page Agent Extension

Description:
Extend the lab agent to handle a multi-step workflow:

1. Search for products on demo site
2. Compare results across 3+ pages
3. Extract best match based on criteria (price, features, availability)
4. Generate a summary report with reasoning

Deliverables:

1. Working code with tests (30 points)
2. Architecture diagram showing data flow (10 points)
3. Security analysis document (15 points)
4. Demo video showing successful run (15 points)
5. README with setup instructions (10 points)
6. Reflection document: challenges faced and solutions (20 points)

Due Date: 1 week post-session
Submission Format: GitHub repository link via course portal

Evaluation Rubric

Code Quality (30 points)

- Code runs without errors (10 pts)
- Proper error handling and logging (8 pts)
- Modular design with clear separation of concerns (7 pts)
- Includes unit tests with >70% coverage (5 pts)

Functionality (25 points)

- Successfully completes multi-page workflow (15 pts)
- LLM integration produces sensible decisions (10 pts)

Security (15 points)

- Implements action allowlist (5 pts)
- Proper API key management (5 pts)
- Security analysis identifies at least 3 real risks with mitigations (5 pts)

Documentation (20 points)

- README is clear and complete (8 pts)
- Architecture diagram accurately represents system (7 pts)
- Reflection shows critical thinking (5 pts)

Demo (10 points)

- Video clearly shows agent working (5 pts)
- Narration explains key decisions (5 pts)

Security & Privacy Considerations

Identified Threats

1. API Key Exposure

- **Impact:** Unauthorized API usage, cost overruns, data breaches
- **Mitigation:** Use environment variables, secrets managers (Azure Key Vault, AWS Secrets Manager), rotate keys regularly, implement usage quotas

2. Malicious Content Injection (SesameOp-style)

- **Impact:** Agent executes unintended commands, data exfiltration, backdoor installation

- **Mitigation:** Sanitize all page content before passing to LLM, use strict output parsing (Pydantic), implement action allowlists, monitor for anomalous patterns
- **Citation:** Microsoft DART, "SesameOp: Backdoor via Assistant API Abuse", November 2025

3. Unintended Actions

- **Impact:** Agent performs destructive operations (delete data, make purchases, modify systems)
- **Mitigation:** Implement human-in-the-loop for critical actions, require confirmation for write operations, maintain detailed audit logs, use read-only modes during testing

4. Data Privacy Violations

- **Impact:** Leaking PII, violating GDPR/CCPA, exposing confidential business data
- **Mitigation:** Minimize data collection, implement data retention policies, use on-device models for sensitive data, encrypt data at rest and in transit, respect robots.txt and terms of service

5. Resource Exhaustion

- **Impact:** Infinite loops, API rate limit hits, excessive cloud costs
- **Mitigation:** Implement timeouts, circuit breakers, rate limiting, cost monitoring alerts, set maximum iteration counts in agentic loops

Security Hardening Checklist

- ☐ All API keys stored in secrets manager, not code
- ☐ Action allowlist implemented and tested
- ☐ Input sanitization for all external content
- ☐ Output validation using Pydantic schemas
- ☐ Rate limiting configured (both API and browser)
- ☐ Comprehensive logging with anomaly detection
- ☐ Human-in-the-loop for critical actions
- ☐ Network egress filtering (allow only approved domains)
- ☐ Regular security audits of agent behavior
- ☐ Incident response plan documented

Deployment Patterns

1. Server-Side Agent

Pros: Full control, powerful hardware, easier debugging
Cons: Requires infrastructure, latency for remote browsers, scaling complexity
Best For: Production systems, heavy workloads, teams with DevOps resources

2. Containerized (Docker/Kubernetes)

Pros: Reproducible, scalable, isolated environments
Cons: Browser in container requires special setup (headless Chrome), resource overhead
Best For: Cloud deployments, CI/CD pipelines, microservices architectures

3. Hybrid (Cloud Browser + Local Logic)

Pros: Reduced latency for users, privacy for sensitive logic
Cons: Complex architecture, state synchronization challenges
Best For: SaaS products, user-facing automation tools

4. In-Browser (WebLLM)

Pros: Maximum privacy, no backend needed, offline capable
Cons: Limited model capability, slower inference, only works in modern browsers
Best For: Privacy-sensitive apps, edge computing, offline tools

---

Further Resources

1. **WebLLM / mlc-ai** - In-browser LLM inference (github.com/mlc-ai/web-llm)
2. **OpenAI AgentKit documentation** - Latest patterns for agentic workflows (November 2025)
3. **Microsoft DART SesameOp report** - Security case study on assistant API abuse
4. **Google LLM Inference for Web** - On-device model integration (ai.google.dev)
5. **Playwright documentation** - Comprehensive guides and API reference (playwright.dev)
6. **LangChain agents** - Framework patterns and examples
7. **Research papers** - ReAct, Toolformer, WebGPT (foundational agent architectures)
8. **Communities** - r/LangChain, Playwright Discord, OpenAI developer forums.