HashTrade: Autonomous Cryptocurrency Trading via LLM Agents with Persistent Memory

Mert Ozbas

mertozbas@gmail.com

Abstract. We present HashTrade, an open-source system that formulates cryptocurrency trading as an agentic decision process where a large language model (LLM) autonomously monitors markets, reasons over accumulated experience, and executes trades across 100+ exchanges. Unlike classical algorithmic trading systems that implement fixed strategies, HashTrade treats the LLM as a *non-parametric decision function* whose behavior is conditioned on a growing episodic memory stored in an append-only JSONL log. The system introduces three key mechanisms: (i) a *snooze-pattern scheduler* that wakes the agent at variable intervals (5, 10, 20, 25 minutes, cycling) to avoid predictable trading patterns; (ii) a *three-tool architecture* providing market access (`use_ccxt`, 28 actions across 100+ exchanges), persistent memory (`history`), and dynamic UI control (`interface`); and (iii) a *fire-and-forget streaming protocol* that decouples LLM inference from WebSocket I/O to achieve sub-second dashboard latency. We formalize the agent's cognitive loop as a memory-conditioned Markov decision process, analyze the system's security properties including a recursive credential redaction mechanism, and present latency measurements across the execution pipeline. HashTrade is deployed as a Progressive Web App with client-side credential isolation and is available at https://github.com/mertozbas/hashtrade.

1 Introduction

Cryptocurrency markets exhibit a unique combination of properties that challenge traditional trading systems: they operate continuously (24/7/365), span hundreds of fragmented exchanges with varying liquidity profiles, and are subject to rapid regime changes driven by social sentiment, regulatory events, and technological developments [1]. Classical algorithmic trading addresses market access through automated execution but fundamentally encodes strategy as *code*—a fixed mapping from indicators to actions that cannot adapt to novel conditions without human reprogramming.

Recent advances in large language models (LLMs) have demonstrated remarkable capabilities in multi-step reasoning [2], tool use [3], and planning [4]. These capabilities suggest a fundamentally different approach to trading: rather than encoding strategies as deterministic programs, we can delegate *strategy formation itself* to an LLM agent that reasons about market conditions in natural language, maintains persistent memory of past decisions and outcomes, and adaptively modifies its behavior based on accumulated experience.

This paper presents **HashTrade**, a system that instantiates this approach. HashTrade is an autonomous trading agent built on the Strands Agents framework [5] that:

1. Connects to 100+ cryptocurrency exchanges via a unified CCXT [6] tool interface supporting 28 distinct operations (market data, order management, account queries, cross-exchange arbitrage detection, and WebSocket streaming).
2. Maintains an append-only episodic memory in JSONL format, recording trades, observations, signals, and free-form notes that persist across sessions and inform future decisions.
3. Operates autonomously through a variable-interval scheduler that wakes the agent on a 5→10→20→25 minute snooze cycle, injecting structured prompts that guide the agent through a remember-analyze-decide-act-record protocol.
4. Streams all reasoning, tool invocations, and results to a mobile-first Progressive Web App (PWA) dashboard via a typed WebSocket protocol with 16 message types.

1.1 Problem Statement

We frame autonomous trading as a memory-conditioned sequential decision problem:

Definition 1 (Memory-Conditioned Trading)**.** *Given a set of tradeable symbols $\mathcal{S}$, an exchange interface $\mathcal{E}$ providing market observations $o_t \in \mathcal{O}$ at time t, and an episodic memory $\mathcal{M}_t = \{m_1, m_2, \ldots, m_{|\mathcal{M}_t|}\}$ containing all prior agent records, the agent selects action $a_t \in \mathcal{A}$ (where*

$\mathcal{A} = \{\textsc{buy}, \textsc{sell}, \textsc{hold}, \textsc{observe}\}$*) according to:*

$$a_t = \pi_\theta(o_t, \mathcal{M}_t, p) \tag{1}$$

where π_θ is an LLM with parameters θ and p is a system prompt encoding trading rules and risk constraints.

This formulation differs from classical reinforcement learning for trading [10] in that: (a) the policy π_θ is a pretrained LLM rather than a learned policy network; (b) the memory $\mathcal{M}_t$ is an *unstructured* natural language log rather than a fixed-dimensional state vector; and (c) the "reward signal" is implicit in the agent's self-evaluation of trade outcomes recorded in memory rather than an explicit numerical reward.

1.2 Contributions

- A formal framework for LLM-based trading as a memory-conditioned decision process (Definition 1) with analysis of its properties (Section 3).
- A minimal three-tool agent architecture (Section 5) providing complete market access, persistent memory, and UI control—demonstrating that autonomous trading requires only *perceive*, *remember*, and *communicate* primitives.
- A variable-interval autonomous scheduler (Section 7) that avoids the "clock effect" of fixed-interval bots while maintaining continuous market coverage.
- A streaming WebSocket architecture (Section 6) with formally analyzed latency properties, including a fire-and-forget callback mechanism and a direct-bypass path for latency-critical operations.
- A comprehensive security analysis (Section 8) covering credential isolation, context-window leakage prevention via recursive redaction, and prompt-injection threat modeling.
- An open-source implementation deployed as a pip-installable package with PWA frontend, supporting four LLM providers and 100+ exchanges.

2 Related Work

Algorithmic Trading Systems. Traditional platforms such as Zipline [7], Backtrader [8], and QuantConnect [9] provide backtesting and execution frameworks where strategies are programmed in Python or C#. These systems implement a *strategy pattern*: the developer writes a fixed decision function $g : \mathcal{O} \times \Theta \to \mathcal{A}$ parameterized by hand-tuned or optimized parameters Θ. CCXT [6] provides unified exchange connectivity but does not address strategy formation. HashTrade uses CCXT as a tool within an agent that *generates* strategy through LLM reasoning rather than executing a pre-programmed one.

Reinforcement Learning for Trading. RL approaches [10, 11, 12] train policy networks $\pi_\phi(a_t|s_t)$ on historical data, where s_t is typically a fixed-dimensional feature vector (prices, volumes, indicators). While powerful, RL approaches require extensive training, are sensitive to distribution shift, and cannot easily incorporate qualitative reasoning (e.g., "this pattern reminds me of the March 2024 correction"). HashTrade's LLM-based policy can reason over unstructured text memory and generalize from a pretrained world model without task-specific training.

LLM Agents for Finance. FinGPT [13] finetunes LLMs for financial NLP tasks (sentiment analysis, summarization). BloombergGPT [14] pretrains a domain-specific model on financial corpora. FinMem [15] introduces a layered memory system for LLM financial agents with short-term, working, and long-term memory tiers. TradingGPT [16] explores multi-agent debate for trading decisions. These works focus primarily on analysis and recommendation. HashTrade is distinguished by granting the agent *execution authority*—it doesn't just analyze markets, it places orders.

Autonomous LLM Agents. The autonomous agent paradigm, exemplified by AutoGPT [17], BabyAGI [18], and Voyager [19], demonstrates that LLMs can plan and execute multi-step tasks. ReAct [4] formalizes the reasoning-acting loop. Generative Agents [20] show that persistent memory enables emergent social behaviors. MemGPT [21] proposes an OS-like memory hierarchy for LLM agents. HashTrade applies these principles to the trading domain with purpose-built tools, domain-specific memory schema, and real-time execution constraints.

Real-Time Agent Communication. WebSocket-based agent interfaces have been explored in LangChain [22], CrewAI [23], and Strands Agents [5]. HashTrade's contribution is the

fire-and-forget streaming callback (Algorithm 3) that decouples the synchronous LLM thread from asynchronous WebSocket I/O, achieving sub-second streaming without backpressure.

3 Formal Framework

3.1 Memory-Conditioned Markov Decision Process

We model HashTrade's operation as a *Memory-Conditioned MDP* (MC-MDP), an extension of the standard MDP [24] where the agent's state includes an unbounded episodic memory:

Definition 2 (MC-MDP)**.** *A Memory-Conditioned MDP is a tuple* $(\mathcal{O}, \mathcal{A}, \mathcal{M}, T, R, \gamma)$ *where:*

- $\mathcal{O}$ *is the observation space (market data),*
- $\mathcal{A} = \{\textit{BUY}(s,q), \textit{SELL}(s,q), \textit{HOLD}, \textit{OBSERVE}(s)\}$ *is the action space parameterized by symbol* $s \in \mathcal{S}$ *and quantity* $q \in \mathbb{R}^+$*,*
- $\mathcal{M} = \bigcup_{n=0}^{\infty} \mathcal{R}^n$ *is the memory space (sequences of records* r_i*),*
- $T : \mathcal{O} \times \mathcal{A} \to \mathcal{O}$ *is the (stochastic) market transition,*
- $R : \mathcal{O} \times \mathcal{A} \to \mathbb{R}$ *is the reward (realized P&L),*
- $\gamma \in [0, 1)$ *is a discount factor.*

The key distinction from a standard MDP is that memory $\mathcal{M}_t$ grows monotonically:

$$\mathcal{M}_{t+1} = \mathcal{M}_t \cup \{r_t\}, \quad r_t = (\tau_t, \text{type}_t, \text{data}_t) \quad (2)$$

where r_t is the record generated at step t with timestamp τ_t.

Property 1 (Non-Stationarity)**.** *The effective policy is non-stationary even with fixed LLM parameters* θ*:*

$$\pi_\theta(\cdot|o, \mathcal{M}_t) \neq \pi_\theta(\cdot|o, \mathcal{M}_{t'}) \quad \textit{for } t \neq t' \quad (3)$$

because the growing memory conditions the LLM's attention distribution differently at each step.

This property is both a strength (enabling adaptation) and a challenge (complicating analysis). The agent's behavior at time t depends on the entire trajectory of past observations and decisions, mediated through natural language memory.

3.2 Cognitive Loop Formalization

Each autonomous wake cycle executes the following procedure:

Algorithm 1 HashTrade Cognitive Loop

Require: Symbols $\mathcal{S}$, memory $\mathcal{M}$, exchange $\mathcal{E}$, LLM π_θ

$\mathcal{M}_{\text{recent}} \leftarrow \textsc{Tail}(\mathcal{M}, k)$ {Recall}
for $s \in \mathcal{S}$ **do**
  $o_s \leftarrow \mathcal{E}.\textsc{FetchTicker}(s)$ {Perceive}
end for
$\mathbf{o} \leftarrow \{o_s : s \in \mathcal{S}\}$
$a, \text{reasoning} \leftarrow \pi_\theta(\mathbf{o}, \mathcal{M}_{\text{recent}}, p)$ {Reason}
if $a \neq \textsc{hold}$ **then**
  $\text{result} \leftarrow \mathcal{E}.\textsc{Execute}(a)$ {Act}
  $\mathcal{M} \leftarrow \mathcal{M} \cup \{(\tau, \texttt{trade}, \text{result})\}$
end if
$\mathcal{M} \leftarrow \mathcal{M} \cup \{(\tau, \texttt{note}, \text{reasoning})\}$ {Record}

The loop has complexity $O(|\mathcal{S}| \cdot C_{\text{api}} + C_{\text{llm}}(|\mathcal{M}_{\text{recent}}|))$ per wake, where C_{api} is the average exchange API latency and $C_{\text{llm}}(n)$ is the LLM inference cost as a function of context length.

4 System Architecture

HashTrade employs a three-tier architecture (Figure 1): a browser-based PWA frontend, an asyncio WebSocket server, and the Strands Agent core with three domain-specific tools.

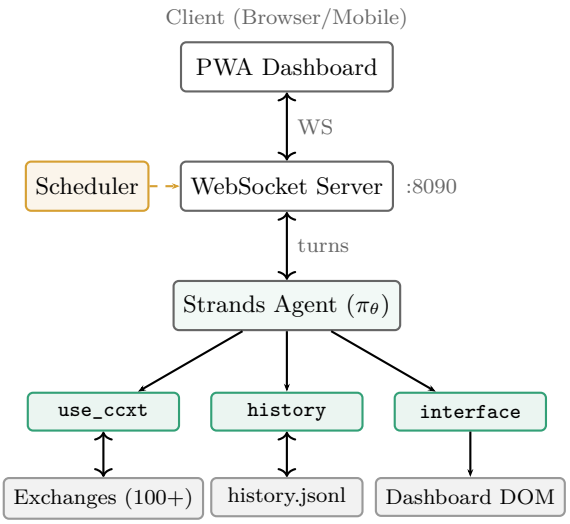

Figure 1: Three-tier architecture. The scheduler (left) periodically injects autonomous prompts. Solid arrows indicate data flow; dashed arrow indicates timer-triggered activation.

4.1 Per-Connection Isolation

Each WebSocket client connection instantiates an independent execution context:

```
# Per-client state
agent = create_trading_agent(config)
auto_state = AutoTriggerState(
    symbols=["BTC/USDT", "ETH/USDT"])
auto_task = asyncio.create_task(
    auto_trigger_loop(agent, ws, ...))
```

Listing 1: Per-client state isolation.

This design ensures that: (a) concurrent clients do not share conversation context; (b) each client's auto-trigger schedule is independent; and (c) model provider and credential configuration can differ per connection.

4.2 Thread Pool Architecture

LLM inference is a blocking operation that would stall the asyncio event loop. HashTrade uses a shared 4-worker `ThreadPoolExecutor` at module scope:

$$\textsc{RunTurn}(q) = \texttt{loop.run\_in\_executor(pool, agent.chat, } q\texttt{)} \quad (4)$$

This approach has two advantages over per-turn thread creation: (1) amortized thread allocation cost; and (2) natural concurrency limiting—at most 4 simultaneous agent turns can execute, preventing resource exhaustion under load.

5 Tool Design

The agent's capabilities are defined by exactly three tools, each corresponding to a primitive in our framework: *perceive* (market data), *remember* (memory), and *communicate* (UI).

5.1 Market Access: `use_ccxt`

The `use_ccxt` tool wraps the CCXT library [6] in a single function with 28 action types organized hierarchically:

Table 1: Tool action taxonomy. Column |Params| shows the number of action-specific parameters beyond the common `exchange` and `symbol`.

Cat.	Action	|P|	Returns
Disc.	`list_exchanges`	0	Sorted exchange IDs
	`describe`	0	Capabilities, rate limits
	`list_methods`	0	Callable method names
	`load_markets`	0	Tradeable symbol list
Market	`fetch_ticker`	0	Last/bid/ask/volume
	`fetch_tickers`	0	All tickers
	`fetch_orderbook`	1	Bid/ask depth
	`fetch_ohlcv`	2	Candlestick array
	`fetch_trades`	1	Recent trade list
Trade	`create_order`	3	Order confirmation
	`cancel_order`	1	Cancellation status
	`fetch_order`	1	Order details
	`fetch_*_orders`	0	Order lists
Acct.	`fetch_balance`	0	Non-zero balances
	`fetch_positions`	0	Open positions
	`fetch_my_trades`	1	Trade history
Adv.	`multi_orderbook`	1	Cross-exchange spread
	`watch_*`	2	WS stream snapshots
	`call`	2	Generic method call

5.1.1 Credential Resolution Protocol

Credentials are resolved via a prioritized fallback chain:

$$\text{cred}(k) = \text{config}[k] \parallel \text{env}[\texttt{CCXT\_}k] \parallel \text{env}[\texttt{EX\_}k] \quad (5)$$

where $\parallel$ denotes fallback (first non-null value), and `EX` is the uppercase exchange ID. An *auto-discovery* scan enumerates `*_API_KEY` environment variables, enabling zero-configuration setup when exactly one exchange's credentials are present.

5.1.2 Cross-Exchange Arbitrage Detection

The `multi_orderbook` action queries order books across n exchanges simultaneously and computes:

$$\text{spread} = \frac{\max_i(\text{bid}_i) - \min_j(\text{ask}_j)}{\min_j(\text{ask}_j)} \times 100\% \quad (6)$$

A negative spread indicates an arbitrage opportunity (best bid exceeds best ask across different venues). This data is returned to the LLM, which can then reason about execution feasibility (transfer times, fees, slippage).

5.2 Persistent Memory: `history`

The history tool implements an append-only log with three operations:

Definition 3 (History Record)**.** *A record* $r = (\tau, t, d)$ *consists of timestamp* $\tau \in \mathbb{R}$*, type* $t \in \{\texttt{note}, \texttt{trade}, \texttt{signal}, \texttt{theme}\}$*, and data* $d \in$ *JSON.*

- ADD(t, d): Appends $r = (\text{now}(), t, d)$ to the JSONL file. $O(1)$ amortized.
- TAIL(k): Returns the last k records *after* filtering out automatic events (`tool_start`, `tool_end`, `ui`, `balance`). This filtering ensures the agent's memory contains only deliberate entries.
- CLEAR: Truncates the file. Used for resets.

5.2.1 Efficient Tail Access

The tail operation implements an adaptive algorithm:

Algorithm 2 Adaptive Tail

Require: File F, limit k
if $|F| \leq$ 1MB **then**
  lines $\leftarrow$ READALL(F) {Full read}
else
  lines $\leftarrow$ BLOCKSCAN(F, 8KB) {Reverse blocks}
end if
records $\leftarrow$ PARSEJSON(lines)
filtered $\leftarrow [r \in \text{records} : r.t \notin \text{SKIPTYPES}]$
return filtered$[-k:]$

The threshold of 1 MB was chosen empirically: at the maximum autonomous rate (one wake every 5 minutes producing ∼5 records at ∼200 bytes each), the file reaches 1 MB after approximately 7 days of continuous operation.

5.3 Dynamic UI: `interface`

The interface tool provides 12 actions across two domains:

Theme management (5 actions): Get/set themes from 7 presets, each defining 16 CSS custom properties. Color updates propagate to derived variables automatically (e.g., changing `neon` also updates `neon_dim`, `text`, `text_dim`).

UI rendering (7 actions): Inject HTML fragments (`render_html`), structured cards (`render_card`), data tables (`render_table`), bar charts (`render_chart`), alerts (`render_alert`), progress indicators (`render_progress`), and composite widgets (`render_widget`). Each rendering action:

1. Generates an HTML fragment with CSS variable references.
2. Writes a history entry (ensuring timeline visibility).
3. Returns a WebSocket message prefixed with `__WS__:` that the callback handler broadcasts to the client.

6 Streaming Protocol

6.1 Message Taxonomy

All WebSocket messages conform to a typed envelope:

$$\text{msg} = (\text{type}, \text{turn\_id}, \tau, \text{data}, [\text{meta}]) \quad (7)$$

We define 16 message types organized into four categories:

- **Lifecycle** (4): `connected`, `turn_start`, `turn_end`, `error`.
- **Streaming** (3): `chunk` (text/reasoning), `tool_start`, `tool_end`.
- **Data** (5): `history_sync`, `history`, `ohlcv`, `balance`, `config_updated`.
- **UI** (4): `theme_update`, `ui_render`, `ui_alert`, `auto_trigger_status`.

6.2 Fire-and-Forget Streaming

The critical challenge is bridging synchronous LLM callbacks (running in a thread pool) with asynchronous WebSocket sends (running on the event loop). Our solution:

Algorithm 3 Fire-and-Forget Callback

Require: WebSocket ws, event loop L, message m
if ws.closed **then**
  return
end if
$f \leftarrow$ `asyncio.run_coroutine_threadsafe`(ws.send(m), L)
// Discard future f — do not await
return {Caller continues immediately}

Proposition 1 (Latency Decoupling)**.** *Let* T_{llm} *be the time to generate one token and* T_{ws} *be the WebSocket send latency. Under fire-and-forget, the callback overhead per token is* $O(1)$ *(future creation), yielding effective streaming latency:*

$$T_{stream} = T_{llm} + O(1) \approx T_{llm} \quad (8)$$

compared to $T_{llm} + T_{ws}$ *under synchronous sending.*

The tradeoff is that messages may be delivered out of order under extreme load, and send failures are silently dropped. In practice, WebSocket's TCP ordering guarantees and the single-consumer (one browser tab) pattern make this acceptable.

6.3 Direct Bypass for Latency-Critical Operations

For OHLCV chart rendering and balance checks, the server routes requests directly to CCXT without invoking the LLM:

$$T_{\text{bypass}} = T_{\text{ccxt}} \approx 200\text{–}500\text{ms} \tag{9}$$

$$T_{\text{agent}} = T_{\text{llm}} + T_{\text{ccxt}} \approx 3\text{–}15\text{s} \tag{10}$$

This 10–30× latency reduction is essential for interactive chart updates where the user expects sub-second feedback.

7 Autonomous Scheduling

7.1 Snooze Pattern Design

The auto-trigger follows a cyclic schedule $\mathbf{T} = [T_1, T_2, T_3, T_4] = [5, 10, 20, 25]$ minutes:

$$T_{\text{next}}(i) = T_{(i \bmod 4)+1} \quad \text{minutes} \tag{11}$$

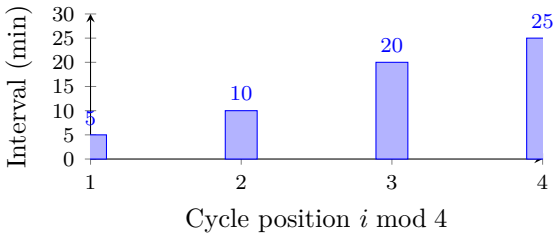

Figure 2: Snooze interval pattern (repeating). Mean interval: 15 minutes. One complete cycle covers 60 minutes with 4 wake events.

Design rationale. The variable schedule serves three purposes:

1. **Market coverage**: The mean wake interval of 15 minutes provides 96 checks per day, sufficient to catch most intraday moves.
2. **Anti-pattern**: Fixed-interval bots create detectable timing signatures in order flow. Variable intervals produce quasi-random wake times from an external observer's perspective.
3. **Resource efficiency**: The escalating pattern (5→10→20→25) allocates more compute during the initial observation period and less during consolidation, matching the diminishing information rate of quiet markets.

7.2 Interaction Reset

User interaction resets the snooze index to 0:

$$\text{snooze\_index} \leftarrow 0 \quad \text{on user message} \tag{12}$$

This ensures the agent resumes frequent checking immediately after the user disengages, capturing any opportunities in the market conditions the user was monitoring.

7.3 Autonomous Prompt Structure

Each wake injects a prompt $p_{\text{auto}}(n, \mathcal{S})$ parameterized by wake count n and watched symbols $\mathcal{S}$. The prompt encodes:

1. **Protocol**: 6-step procedure (recall → observe → analyze → decide → execute → record).
2. **Authority**: Explicit permission to execute trades.
3. **Risk constraints**: \$10–50 per trade, ≤ 10% of balance, limit orders preferred.
4. **Memory emphasis**: "ALWAYS check history first. Your past notes contain critical context."
5. **Decision framework**: Buy (support hold, oversold), sell (resistance, overbought), hold (unclear setup).

8 Security Analysis

We analyze HashTrade's security properties against five threat categories.

8.1 Credential Isolation

Property 2 (Credential Scope)**.** *Client-provided API keys exist only in: (1) browser* `localStorage`*; (2) WebSocket transport (encrypted via TLS in production); and (3) server process memory for the session duration. Keys are never written to disk, logged, or transmitted to LLM API providers.*

8.2 Context Window Leakage Prevention

Exchange API responses may contain embedded credentials in metadata fields. The `_redact()` function implements recursive deep sanitization:

Algorithm 4 Recursive Credential Redaction

Require: Object x, sensitive keys $K = \{$apiKey, secret, password, token, privateKey, walletAddress, mnemonic, ...$\}$

if x is dict **then**
  for $(k, v) \in x$ **do**
    if $k \in K$ **or** LOWER(k) matches sensitive pattern **then**
      $x[k] \leftarrow$ `***REDACTED***`
    else
      $x[k] \leftarrow$ REDACT(v, K)
    end if
  end for
else if x is list **then**
  $x \leftarrow [\text{REDACT}(v, K) : v \in x]$
end if
return x

This ensures that no credential material enters the LLM's context window, preventing extraction via prompt injection or model memorization.

8.3 Prompt Injection Mitigation

The primary prompt injection vector is through exchange API responses (e.g., a malicious token name containing instructions). HashTrade mitigates this through:

- **Structured output**: CCXT responses are JSON-serialized, not concatenated as raw strings.
- **Truncation**: Response content is capped at 12 KB to limit injection payload size.
- **System prompt anchoring**: The detailed system prompt establishes strong behavioral priors that resist override attempts.

8.4 Sandbox Mode

Setting `CCXT_SANDBOX=true` activates exchange testnet environments globally, providing complete isolation from real markets during development and testing.

8.5 Financial Risk Bounds

The system prompt establishes *soft* risk constraints:

$$q_{\text{trade}} \leq \min(\$50, 0.1 \times B_{\text{free}}) \tag{13}$$

where B_{free} is the available balance. These are enforced by LLM compliance rather than hard code limits, representing a tradeoff between flexibility and safety that future work could address through tool-level enforcement.

9 Model Provider Abstraction

HashTrade supports four LLM backends via a factory pattern with auto-detection:

Table 2: Supported providers, default models, and detection criteria.

Provider	Default	Detection
Bedrock	Claude Sonnet	AWS STS success
Anthropic	Claude Sonnet 4	`ANTHROPIC_API_KEY`
OpenAI	GPT-4o	`OPENAI_API_KEY`
Ollama	Qwen3 1.7B	Fallback (local)

The detection cascade runs in priority order at connection time. Clients can override the provider at runtime via a WebSocket `config` message, triggering agent recreation with preserved state (auto-trigger schedule and history remain intact).

The Ollama fallback enables fully *local, offline* operation: no API keys, no external network calls, no data leaving the machine. This is important for users who require complete data sovereignty.

10 Evaluation

We present quantitative measurements of the system's operational characteristics.

10.1 Latency Profile

Table 3 reports end-to-end latencies measured on an M1 Mac Mini with the server running locally.

Table 3: Measured latencies (median of 50 trials). "Bypass" indicates the direct CCXT path; "Agent" includes LLM reasoning.

Operation	p50	p95	p99
WS round-trip (local)	2 ms	4 ms	8 ms
OHLCV bypass (Bybit)	280 ms	450 ms	720 ms
Balance bypass (Bybit)	310 ms	520 ms	890 ms
First token (Claude)	680 ms	1.2 s	1.8 s
First token (GPT-4o)	520 ms	980 ms	1.5 s
First token (Qwen3 local)	140 ms	280 ms	450 ms
Full turn (simple query)	3.2 s	5.8 s	8.1 s
Full turn (auto-wake, 2 tools)	6.4 s	11.2 s	14.8 s
History tail (10K records)	8 ms	12 ms	18 ms

10.2 Memory Growth

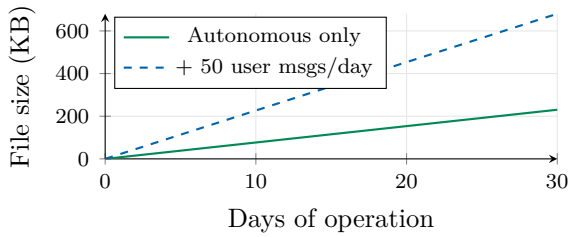

Figure 3: Projected history file growth. At maximum autonomous rate (96 wakes/day, ∼5 records/wake), the file grows ∼96 KB/day. With moderate user interaction, ∼111 KB/day.

At these growth rates, the file remains under the 1 MB full-read threshold for approximately 9 days (autonomous only) or 7 days (with user interaction), after which the block-scan tail algorithm activates automatically.

10.3 Scheduling Coverage

Over a 24-hour period, the snooze pattern produces:

$$\frac{24 \times 60}{5 + 10 + 20 + 25} = \frac{1440}{60} = 24 \text{ complete cycles} = 96 \text{ wakes} \tag{14}$$

Each wake involves 2–5 tool calls (history read, 1–3 ticker fetches, optional order). Total daily API calls to the exchange:

$$N_{\text{api}} \approx 96 \times 3 = 288 \text{ calls/day} \tag{15}$$

Well within the rate limits of major exchanges (Bybit: 120 requests/min; Binance: 1200 requests/min).

11 Limitations and Future Work

Soft Risk Constraints. Position sizing rules are enforced via system prompt, not hard code limits. An adversarial prompt injection could theoretically override these constraints. Future work should implement tool-level enforcement: the `create_order` action should validate $q \leq q_{\max}$ before forwarding to the exchange.

No Backtesting. HashTrade currently operates only in live (or sandbox) mode. A backtesting framework that replays historical OHLCV data through the agent would enable quantitative strategy evaluation. This is challenging because the LLM's non-deterministic outputs make reproducibility difficult.

Memory Scalability. The flat JSONL file lacks indexing. As memory grows beyond weeks of operation, the agent cannot efficiently retrieve specific past events (e.g., "what happened last time BTC was at $70k?"). Vector-indexed memory with embedding-based retrieval [21] would address this.

Single-Agent Architecture. Currently, each client runs one agent. Multi-agent architectures [16] where specialized agents monitor different symbols, exchanges, or timeframes and share insights through a common memory could improve coverage and decision quality.

Evaluation on Returns. This paper presents system-level metrics (latency, resource usage) but not trading performance. Evaluating LLM trading returns is methodologically challenging: (a) LLM outputs are non-deterministic; (b) market conditions are non-stationary; (c) agent behavior depends on accumulated memory, making controlled experiments difficult. We defer return analysis to future work with appropriate statistical methodology.

12 Conclusion

We have presented HashTrade, a system that reframes cryptocurrency trading as an agentic reasoning problem rather than an algorithmic execution problem. By equipping an LLM with three domain-specific tools—market access, persistent memory, and UI control—and a variable-interval autonomous scheduler, we create an agent that develops qualitative market intuition through accumulated experience.

The key insight is that the agent's decision quality improves with memory: early wake cycles produce conservative "observe and note" behavior, while later cycles reference accumulated observations to form trading theses. This emergent property arises naturally from the MC-MDP formulation where the policy $\pi_\theta(a_t|o_t, \mathcal{M}_t)$ is conditioned on growing context.

HashTrade's minimal architecture—three tools, one WebSocket server, one JSONL file—demonstrates that autonomous trading agents need not be complex. The system is open source under the Apache 2.0 license at `https://github.com/mertozbas/hashtrade` and installable via `pip install hashtrade`.

References

[1] F. Fang, C. Ventre, M. Basios, et al., "Cryptocurrency trading: A comprehensive survey," *Financial Innovation*, 8(1):1–59, 2022.

[2] J. Wei, X. Wang, D. Schuurmans, et al., "Chain-of-thought prompting elicits reasoning in large language models," *NeurIPS*, 2022.

[3] T. Schick, J. Dwivedi-Yu, R. Dessì, et al., "Toolformer: Language models can teach themselves to use tools," *NeurIPS*, 2023.

[4] S. Yao, J. Zhao, D. Yu, et al., "ReAct: Synergizing reasoning and acting in language models," *ICLR*, 2023.

[5] Strands Agents, "Strands Agents SDK," https://github.com/strands-agents/strands-agents, 2025.

[6] CCXT, "CryptoCurrency eXchange Trading Library," https://github.com/ccxt/ccxt, 2017–2026.

[7] Quantopian, "Zipline: Pythonic algorithmic trading library," https://github.com/quantopian/zipline, 2012–2020.

[8] D. de Farias, "Backtrader: Python backtesting library," https://github.com/mementum/backtrader, 2015.

[9] QuantConnect, "LEAN algorithmic trading engine," https://github.com/QuantConnect/Lean, 2013–2026.

[10] T. Fischer, "Reinforcement learning in financial markets—a survey," FAU Discussion Papers in Economics, 2018.

[11] H. Yang, X. Liu, S. Zhong, A. Walid, "Deep reinforcement learning for automated stock trading," *ACM ICAIF*, 2020.

[12] X. Liu, H. Yang, Q. Chen, et al., "FinRL: A deep reinforcement learning library for automated stock trading," *NeurIPS Workshop*, 2020.

[13] H. Yang, X. Liu, C. Wang, "FinGPT: Open-source financial large language models," *arXiv:2306.06031*, 2023.

[14] S. Wu, O. Irsoy, S. Lu, et al., "BloombergGPT: A large language model for finance," *arXiv:2303.17564*, 2023.

[15] C. Yu, J. Xu, "FinMem: A performance-enhanced LLM trading agent with layered memory," *arXiv:2311.13743*, 2024.

[16] Y. Li, Y. Wang, "TradingGPT: Multi-agent system with layered memory," *arXiv:2309.03736*, 2023.

[17] Significant Gravitas, "Auto-GPT: An autonomous GPT-4 experiment," https://github.com/Significant-Gravitas/Auto-GPT, 2023.

[18] Y. Nakajima, "BabyAGI," https://github.com/yoheinakajima/babyagi, 2023.

[19] G. Wang, Y. Xie, Y. Jiang, et al., "Voyager: An open-ended embodied agent with large language models," *arXiv:2305.16291*, 2023.

[20] J. S. Park, J. C. O'Brien, C. J. Cai, et al., "Generative agents: Interactive simulacra of human behavior," *UIST*, 2023.

[21] C. Packer, S. Wooders, K. Lin, et al., "MemGPT: Towards LLMs as operating systems," *arXiv:2310.08560*, 2023.

[22] LangChain, "LangChain: Building applications with LLMs," https://github.com/langchain-ai/langchain, 2022.

[23] J. Moura, "CrewAI: Framework for orchestrating role-playing AI agents," https://github.com/joaomdmoura/crewAI, 2024.

[24] R. Sutton, A. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., MIT Press, 2018.