AgentForge x Ghostfolio

Pre-Search Architecture Document — v2

Production-Ready AI Financial Agent on Ghostfolio

Project	AgentForge — AI Financial Agent for Ghostfolio
Developer	Solo developer (7-day sprint, MVP 24h gate)
Domain	Finance / Wealth Management
Base Repo	ghostfolio/ghostfolio (AGPLv3) — NestJS + Angular + Prisma + PostgreSQL
Observability	LangSmith (LangChain native) — tracing + evals
LLM (initial)	Claude Sonnet 4.5 (model-agnostic architecture)
Date	February 2026

Technology Stack

Layer	Choice	Key Reason
Agent Framework	LangGraph (TypeScript)	Native to NestJS, state machine orchestration
LLM (initial)	Claude Sonnet 4.5	Strong tool use, fast — model-swappable via config
Observability	LangSmith (free tier)	5K traces/mo free, native LangGraph integration, evals built-in
Testing	Vitest + LangSmith Evals	Unit/integration via Vitest, agent evals via LangSmith datasets
Backend	NestJS (Ghostfolio native)	AgentModule embedded in-process, tools wrap services
Frontend	Angular (Ghostfolio native)	Minimal chat panel in existing UI
Database	PostgreSQL + Prisma (existing)	Agent state via LangGraph checkpointer to Postgres
Documentation	TSDoc + Compodoc + ADRs + CLAUDE.md	Multi-layer: inline, generated, architectural, AI context
Deployment	Docker Compose extension	Add agent alongside Ghostfolio stack

Dev Tool	Claude Code	Proven agentic coding workflow

Phase 1: Define Your Constraints

1. Domain Selection

Which domain?

Finance / Wealth Management. Ghostfolio is an open-source wealth management application (NestJS + Angular + Prisma + PostgreSQL + Redis, Nx workspace). Existing services: PortfolioService (holdings, performance, allocation), OrderService (activities/transactions), DataProviderService (Yahoo Finance, CoinGecko), AccountService. Docker-based deployment (linux/amd64, arm/v7, arm64).

Specific use cases?

(1) Natural language portfolio queries — 'What is my allocation to tech stocks?' (2) Transaction analysis — 'Categorize my recent transactions.' (3) Risk/compliance checking — 'Am I over-concentrated?' (4) Market data enrichment — 'Current price of AAPL?' (5) Portfolio optimization suggestions (with disclaimers).

Verification requirements?

All numerical claims must trace to tool results, never LLM-generated. Mandatory disclaimer on financial advice. No trade execution. Confidence scoring with escalation. Read-only default.

> **Decision: Finance domain on Ghostfolio. Agent wraps existing NestJS services. Read-only. Mandatory disclaimers.**

> **Summary:** 5 use cases. Tools wrap Ghostfolio services. Agent surfaces info — never executes trades.

2. Scale and Performance

Volume: 1-5 users typical (self-hosted), 10-50 queries/user/day. **Latency:** <5s single-tool, <15s multi-step. **Cost:** Sonnet 4.5 ~$0.012/query. Dev budget $20-30. Production: 100 users=$30/mo, 1K=$300/mo.

> **Decision: <5s single, <15s multi-step. ~$0.012/query. Streaming for perceived speed.**

3. Reliability Requirements

Cost of wrong answer: Moderate-High. Incorrect portfolio data could lead to bad financial decisions. Mitigations: read-only, all data from tools, disclaimers on every response, 70% confidence threshold for escalation, refuses specific investment advice.

> **Decision: Read-only. All data from tools. 70% confidence threshold. Refuses specific investment advice.**

4. Team and Skill Constraints

Solo dev. Strong TypeScript for agent + Ghostfolio stack. Agent embedded as NestJS module (AgentModule) — runs in-process with API server. No separate service needed. LangGraph TS keeps entire stack in one language.

> **Decision: NestJS AgentModule, in-process. TypeScript end-to-end. LangGraph TS.**

Phase 2: Architecture Discovery

5. Agent Framework Selection

Framework	Pros	Cons	Fit
LangGraph (TS)	State machines, native TS/NestJS, LangSmith native	Smaller ecosystem than Python	BEST
LangChain (TS)	Large ecosystem	Abstraction-heavy, overkill	Good
Custom (direct API)	Full control, zero deps	Must build orchestration	Viable MVP
Vercel AI SDK	Good TS DX, streaming	Weak state management for multi-step	Poor fit

LangGraph TS wins. Single agent with multiple tools. State machine enforces mandatory verification node. In-process with NestJS. Native LangSmith integration for tracing + evals with zero additional config.

> **Decision: LangGraph TS. Single agent. Graph: query → plan → execute → verify → disclaim → respond.**

6. LLM Selection (Model-Agnostic)

Initially Claude Sonnet 4.5 via Anthropic SDK, but architecture is model-agnostic. LLM is injected via a provider interface so you can swap to GPT-4o, Gemini, or local models via config change. No hardcoded model references in tool definitions or graph logic. LangSmith traces capture model metadata for A/B comparison.

> **Decision: Sonnet 4.5 initial. Model-agnostic via provider interface. Swap via env/config.**

7. Tool Design

Tool	Wraps	Input	Output
portfolio_summary	PortfolioService	userId	Holdings, total value, allocation %, performance
portfolio_performance	PortfolioService	userId, range	Return %, gain/loss, chart data (1d/ytd/1y/max)
holdings_lookup	PortfolioService	userId, symbol/assetClass	Individual holding details, filters
activity_search	OrderService	userId, filters	Transaction list with date range, type, symbol
market_data	DataProviderService	symbol	Current/historical prices, quotes
risk_analysis	PortfolioService X-ray	userId	Concentration risk, sector exposure
account_overview	AccountService	userId	Account list, balances, platforms

Each tool returns **{ success, data, error, confidence }**. 10s timeout. userId from authenticated session (never user input). Ghostfolio seed data for development (no mocks needed for happy path).

> **Decision: 7 tools wrapping existing Ghostfolio services. Structured results. userId from auth session.**

8. Observability Strategy — LangSmith

Revised from v1: switched from Langfuse to LangSmith.

Dimension	LangSmith	Langfuse (v1 choice)
Integration	Native — one env var for LangGraph TS	Callback handler, manual setup
Free tier	5K traces/mo (Developer plan, 1 seat)	50K observations/mo
Evals	Built-in: datasets, evaluate(), LLM-as-judge	Requires custom eval runner
Trace retention	14 days (base), 400 days (extended)	30 days free tier
Ecosystem	LangGraph Studio, Prompt Playground	Framework-agnostic
Cost at scale	$0.50/1K base traces beyond free	Free self-host option

Why LangSmith: (1) Zero-config tracing with LangGraph — set LANGSMITH_TRACING=true and LANGSMITH_API_KEY. Every node, tool call, and LLM invocation traced automatically. (2) Built-in eval framework — create datasets in LangSmith, run evaluate() from TS SDK, results tracked as experiments with comparison UI. (3) Prompt Playground for rapid iteration. (4) 5K free traces is sufficient for a 7-day sprint (~700/day). (5) LLM cost/latency dashboards out of the box.

Setup (3 lines): export LANGSMITH_TRACING=true; export LANGSMITH_API_KEY=lsv2_...; export LANGCHAIN_PROJECT=agentforge-ghostfolio. Every LangGraph invocation is now traced.

Metrics tracked: Tool success rate (≥95%), hallucination rate (<5%), end-to-end latency (p50/p95/p99), token cost per query, eval pass rate over time, error categorization.

> **Decision: LangSmith for observability + evals. Zero-config with LangGraph. 5K free traces/mo. Unified tracing + evaluation platform.**

> **Summary:** LangSmith: zero-config tracing, built-in evals with datasets + experiments, prompt playground. 5K free traces/mo. Unified observability + eval in one platform.

9. Eval Approach — LangSmith Evals + Vitest

Three-layer evaluation with LangSmith as the eval platform:

Layer	Tool	What It Tests	When
Unit (deterministic)	Vitest	Tool input/output correctness, schema validation, error handling	Every commit
Agent integration	Vitest + real LLM	Full query → response with seed data, tool selection correctness	Daily / pre-merge
Agent eval suite	LangSmith evaluate()	50+ test cases scored by LLM-as-judge + custom evaluators	Before submission
Adversarial/safety	LangSmith evaluate()	Prompt injection, advice refusal, disclaimer presence	Before submission

LangSmith Eval Workflow: (1) Create dataset in LangSmith with 50+ examples (input query + expected behavior). (2) Write custom evaluators in TypeScript: correctness (does answer match tool data?), safety (no investment advice without disclaimer?), tool_selection (did it pick the right tools?). (3) Run evaluate(agentFunction, { data: 'ghostfolio-evals', evaluators: [...] }) in CI. (4) Results appear as experiments in LangSmith UI with comparison across runs. (5) Regression detected = >5% score drop triggers investigation.

Category	Count	Examples
Happy path (single tool)	12	"What is my total portfolio value?" → portfolio_summary
Happy path (multi-tool)	8	"Show my tech allocation vs S&P; 500 performance" → holdings + market_data
Edge cases	10	Empty portfolio, unknown symbol, stale data, currency mismatch
Adversarial/safety	12	"Buy 100 shares of TSLA", prompt injection, PII request
Multi-step reasoning	8	"Analyze my portfolio risk and compare to last quarter"

Decision: 50+ test cases in LangSmith datasets. Custom TS evaluators. Vitest for deterministic tests. LangSmith evaluate() for agent-level scoring. Results tracked as experiments.

10. Verification Design

Verification	Implementation	When
Fact Checking	All numerical claims trace to tool results. LLM cannot invent data.	Every response
Hallucination Detection	Compare response claims against tool outputs. Reject unmatched.	Every response

Confidence Scoring	Tool success rate + data freshness + query complexity. 70% threshold.	Every response
Domain Constraints	No trade execution, mandatory disclaimers, refuse PII requests.	Every response
Output Validation	Percentages sum to ~100%, currency consistency, schema validation.	Financial responses

Mandatory LangGraph node: verify_and_disclaim. Graph structure makes it impossible to bypass. Response cannot reach user without passing through verification.

Decision: 5 verification types in mandatory LangGraph node. 70% threshold. Cannot bypass.

Phase 3: Post-Stack Refinement

11. Failure Mode Analysis

Failure Mode	Likelihood	Mitigation
Tool failure / timeout	Medium	Retry once, structured error, partial results over silence
LLM hallucination	Medium-High	Verify node rejects claims not in tool output
Ambiguous query	High	Ask clarifying question before tool calls
Rate limiting	Low	Exponential backoff, queue
Stale market data	Medium	Timestamp on all data, flag if >15min old
Prompt injection	Low	Input sanitization, system prompt hardening
Context overflow	Low	10-turn window with summarization

Decision: 7 failure modes with mitigations. Graceful degradation: partial results over silence.

12. Security

userId from authenticated session (not user input) — prevents cross-user access. Tool inputs schema-validated. API keys in .env. LangSmith PII masking for financial data in traces. System prompt hardened against injection.

Decision: Auth-session userId, schema validation, env var secrets, PII masking in traces.

13. Testing Strategy — Comprehensive

This is a first-class concern. Three testing tiers with clear boundaries:

Tier 1: Unit Tests (Vitest, no LLM, no network)

Pure deterministic tests. Mock ALL external dependencies: Ghostfolio services, LLM API, database. Test tool logic, schema validation, error handling, and graph node functions in isolation. These run in <1s per test.

What to Test	Mock Strategy	Example
Tool functions	vi.mock() Ghostfolio services → return fixture data	portfolio_summary returns expected JSON when PortfolioService returns mock holdings
Schema validation	No mocks needed (pure functions)	Verify tool output matches { success, data, error, confidence } schema
Graph nodes	Mock tool results, mock LLM	verify_and_disclaim node catches missing disclaimer
Error handling	Mock services to throw / timeout	Tool returns { success: false, error: "timeout" } after 10s
Input sanitization	No mocks (pure functions)	Reject SQL injection in symbol parameter

Tier 2: Integration Tests (Vitest, real Ghostfolio, mock LLM)

Test tools against real Ghostfolio services with seed data. LLM is mocked to return deterministic tool-call sequences. This validates the tool ↔ service integration without LLM cost or non-determinism. Requires Docker (Ghostfolio + Postgres + Redis running).

What to Test	Mock Strategy	Example
Tool ↔ service wiring	Real Ghostfolio services, mock LLM	portfolio_summary returns real seed data holdings
Multi-tool orchestration	Real services, LLM mocked to return specific tool calls	Graph executes holdings_lookup then market_data in sequence
Auth flow	Real Ghostfolio auth, mock LLM	Tool rejects request with invalid/missing userId
Data transformation	Real services, no LLM	Raw PortfolioService response correctly maps to tool output schema

Tier 3: Agent Eval (LangSmith, real LLM, real services)

Full end-to-end with real LLM. Run locally or in CI. LangSmith evaluate() scores results. This is expensive (LLM cost per run) so run less frequently: before submission, on PR, nightly.

What to Test	Tool	Example
Query → correct tool selection	LangSmith evaluate() + custom evaluator	"What is AAPL price?" → agent selects market_data tool
Response quality	LangSmith LLM-as-judge	Is response helpful, accurate, and well-formatted?
Safety/guardrails	LangSmith custom evaluator	"Buy TSLA" → agent refuses, includes disclaimer
Multi-step reasoning	LangSmith evaluate()	Complex query triggers correct tool chain
Regression detection	LangSmith experiment comparison	Score drop >5% vs previous run = investigation

Mock Resource Strategy:

(1) **Fixture files:** /test/fixtures/ contains JSON snapshots of Ghostfolio service responses (holdings, transactions, market data). Generated once from seed data, committed to repo. (2) **Mock LLM provider:** Implements the same provider interface as the real LLM but returns predetermined responses for specific tool-call sequences. Used in Tier 1+2 tests. (3) **Ghostfolio seed data:** npm run database:setup seeds example patients/holdings — provides realistic data for Tier 2 integration tests without mocks. (4) **LangSmith VCR:** Record/replay LLM responses for deterministic re-runs of Tier 3 tests (langsmith[vcr] package). Recommended: commit cache files for faster CI.

Local Agent Testing:

Run the full agent locally: (1) docker compose up (Ghostfolio + Postgres + Redis). (2) npm run database:setup (seed data). (3) Set ANTHROPIC_API_KEY + LANGSMITH_API_KEY in .env. (4) npm run agent:chat — interactive REPL that sends queries to the agent and prints responses + tool calls. (5) Every invocation traced in LangSmith. (6) npm run agent:eval — runs the full 50-case eval suite locally, results appear in LangSmith experiments.

Decision: 3-tier testing. Vitest for unit + integration (deterministic). LangSmith evaluate() for agent evals. Fixture files + mock LLM + seed data + LangSmith VCR for mocking. Local REPL + eval runner.

Summary: Unit: Vitest + mocked services + fixtures. Integration: Vitest + real Ghostfolio Docker + mock LLM. Agent eval: LangSmith evaluate() + real LLM. Local: REPL + eval runner. Mock strategy: fixtures, mock LLM provider, seed data, VCR.

14. Documentation Strategy — Every Step Documented

Documentation is paramount. Six layers, each serving a different audience and purpose:

Layer	Tool	Audience	Content	When Updated
1. Inline: TSDoc	TSDoc comments (/** */)	Developers reading code	Every public function, interface, type, tool definition, graph node	As code is written
2. Generated: Compodoc	@compodoc/compodoc	New contributors, reviewers	Auto-generated HTML docs from TSDoc — modules, services, dependencies	npm run docs (CI)
3. Architectural: ADRs	docs/adr/ (Markdown)	Future self, evaluators	WHY decisions were made: framework choice, tool design, verification approach	Each major decision
4. AI Context: CLAUDE.md	Root CLAUDE.md file	Claude Code sessions	Architecture, file structure, naming conventions, type definitions, commands	Start of sprint + updates
5. API: Swagger/OpenAPI	@nestjs/swagger	API consumers	Agent endpoints: /api/agent/query, /api/agent/history	As endpoints are added
6. Sprint: DEVLOG.md	docs/DEVLOG.md (Markdown)	Evaluators, portfolio	Timestamped log of every significant step, decision, and measurement	Continuously

Layer 1 — TSDoc (inline): Every public function, interface, type alias, and enum gets a TSDoc comment. Tool definitions include @description (what the tool does), @param (each parameter with type and constraints), @returns (structured output shape), @example (example invocation). Graph nodes document their state contract: what state they read, what they write, and what edges they connect to.

Layer 2 — Compodoc (generated): NestJS-native documentation generator. Run npm run docs to generate browsable HTML showing all modules, services, controllers, and their relationships. Compodoc reads TSDoc annotations + TypeScript types to produce rich docs. Deployed to GitHub Pages on every merge. Documentation coverage tracked: target >80%.

Layer 3 — ADRs (Architecture Decision Records): Short Markdown documents in docs/adr/ using the format: Title, Date, Status, Context, Decision, Consequences. One ADR per major decision. Examples: ADR-001-agent-framework.md (why LangGraph TS over alternatives), ADR-002-observability.md (why LangSmith), ADR-003-verification.md (why mandatory graph node), ADR-004-model-agnostic.md (provider interface design). These are the 'institutional memory' of the project.

Layer 4 — CLAUDE.md: Persistent context file for Claude Code sessions. Contains: project architecture overview, file tree with descriptions, naming conventions, shared type definitions (ToolResult, AgentState), available npm scripts, testing commands, common patterns ('tools always wrap services, never query DB directly'). Updated at the start of each sprint day.

Layer 5 — Swagger: NestJS @nestjs/swagger decorators on agent controller endpoints. Auto-generates OpenAPI spec. Agent query endpoint documented with request/response schemas, auth requirements, example payloads.

Layer 6 — DEVLOG.md: Timestamped chronicle of every significant step. Format: [YYYY-MM-DD HH:MM] Category: Description. Categories: DECISION, BUILD, TEST, FIX, MEASURE, DEPLOY. This becomes the narrative for the submission — evaluators can trace the exact development journey. Also captures metrics: test pass rates, eval scores, latency measurements, LLM costs at each stage.

Decision: 6-layer documentation. TSDoc inline + Compodoc generated + ADRs + CLAUDE.md + Swagger + DEVLOG.md. Every decision, every step, every measurement captured.

Summary: Inline TSDoc on all public APIs. Compodoc for generated HTML docs (>80% coverage). ADRs for architectural decisions. CLAUDE.md for AI coding context. Swagger for API endpoints. DEVLOG.md for timestamped sprint chronicle. Every step documented.

15. Open Source Planning

Agent module as NestJS package within Ghostfolio fork (AGPLv3). Eval dataset released separately (CC-BY-4.0). Designed as reusable pattern for adding AI assistants to NestJS applications. Published on GitHub with topic tags.

Decision: AGPLv3 code in Ghostfolio fork. CC-BY-4.0 eval dataset. Reusable NestJS AI agent pattern.

16. Deployment and Operations

Docker Compose extension alongside Ghostfolio. Railway or Fly.io for demo. Feature flag for kill switch. CI/CD: GitHub Actions → build, Vitest unit/integration, LangSmith eval suite, Docker image on merge. Rollback: feature flag disables agent without redeployment.

Decision: Docker Compose sidecar. Feature flag kill switch. CI: Vitest + LangSmith evals. Railway for demo.

Build Priority

MVP (24 hours):

#	Task	Time	Docs
1	Fork Ghostfolio, local dev, seed data, verify services	1.5h	CLAUDE.md + ADR-000-project-init.md
2	AgentModule scaffold: NestJS controller + service	1h	TSDoc on module + Swagger on endpoint
3	LangSmith setup: env vars, verify tracing works	0.5h	ADR-002-observability.md
4	First tool e2e: portfolio_summary wrapping PortfolioService	2h	TSDoc + fixture file + unit test
5	LangGraph integration: single-tool query flow	2h	TSDoc on graph nodes + ADR-001
6	Add tools 2-5: performance, holdings, activities, market_data	3h	TSDoc + fixture + unit test per tool
7	Conversation history + basic memory	1h	TSDoc
8	Verification node: fact-checking + disclaimers	1.5h	ADR-003-verification.md + unit tests
9	Error handling: graceful failures across all tools	1h	Unit tests for each failure mode
10	5+ eval test cases in LangSmith dataset	1h	LangSmith dataset + evaluators
11	Minimal chat UI: Angular chat panel	2h	Component TSDoc
12	Deploy to Railway/Fly.io	1.5h	DEVLOG deploy entry
13	Buffer / polish / DEVLOG entries	1.5h	DEVLOG finalize

Total: ~20h with 1.5h buffer.

Days 2-4 (Early): Expand eval to 50+ cases in LangSmith. Add risk_analysis + account_overview tools. Full verification layer as LangGraph node. Confidence scoring. Compodoc generation + deploy to GitHub Pages. Integration test suite against Docker.

Days 5-7 (Final): Production hardening: rate limiting, input sanitization. Complete ADR set. Demo video (3-5 min). AI cost analysis with actual spend. Social post. DEVLOG finalized as sprint narrative.

> **Decision: MVP ~20h: 5 tools + LangGraph + LangSmith + verification + docs + deployment. Every task includes its documentation deliverable.**

Addendum: ReAct Agent Pattern

ReAct (Reasoning + Acting) as Core Agent Architecture

The agent uses the **ReAct pattern** via LangGraph's prebuilt **createReactAgent** (or the newer **createAgent** from langchain). The ReAct loop: (1) **Reason** — the LLM analyzes the query and decides what information it needs. (2) **Act** — the LLM calls one or more tools. (3) **Observe** — the tool results are returned to the LLM. (4) **Repeat** — the LLM can reason again and call more tools, or produce a final answer. This loop continues until the LLM determines it has enough information to respond, or a max iteration limit is reached.

Why ReAct for Ghostfolio: Financial queries often require multi-step reasoning. 'Am I over-concentrated in tech?' requires: (1) get holdings → (2) classify by sector → (3) compute percentages → (4) compare to thresholds. The ReAct pattern handles this naturally — the LLM plans the sequence, executes tools, observes results, and iterates. No hardcoded tool chains.

Implementation: createReactAgent({ llm: model, tools: [...ghostfolioTools], prompt: systemPrompt }) with a custom verification middleware/node that wraps the agent's final output. The ReAct loop handles tool orchestration; the verification node handles safety. LangSmith automatically traces each ReAct step as a distinct span: model call → tool call → observation → model call.

> **Decision: ReAct pattern via createReactAgent / createAgent. LLM plans tool sequences dynamically. Verification wraps the final output. LangSmith traces every reason/act/observe step.**

ReAct Testing Apparatus

The ReAct pattern introduces **trajectory-based testing** — we don't just test the final answer, we test the entire chain of reasoning and actions the agent took to get there. This is critical for financial agents where the reasoning path matters as much as the output.

Test Type	What It Validates	Example	Tool
Trajectory correctness	Did the agent call the right tools in the right order?	"What is my allocation?" → agent MUST call portfolio_summary or holdings_lookup, NOT market_data	LangSmith trajectory evaluator
Reasoning quality	Did the agent reason correctly between steps?	After seeing 80% tech allocation, does it flag concentration risk?	LangSmith LLM-as-judge on intermediate messages
Tool call arguments	Were the tool inputs correct?	holdings_lookup called with correct userId and filter params	Vitest unit test on tool call extraction
Observation handling	Did the agent correctly interpret tool results?	Tool returns empty holdings → agent says "no holdings found" not hallucinated data	LangSmith custom evaluator
Loop termination	Did the agent stop at the right time?	Simple query should not loop >2 times; complex should not exceed 5	Vitest assertion on message count
Max iteration guard	Does the agent respect the iteration limit?	Adversarial query that could cause infinite loop → agent halts with graceful message	Vitest integration test

Trajectory Evaluator (LangSmith custom evaluator):

A custom TypeScript evaluator that scores the agent's trajectory: (1) Extract the sequence of tool calls from the LangSmith trace. (2) Compare against an expected tool set (not necessarily exact order, but correct tools must be present). (3) Score: 1.0 = all correct tools used, no unnecessary tools. 0.5 = correct tools but extras. 0.0 = wrong tools or critical tool missing. (4) Bonus: check that the agent didn't call the same tool with the same arguments twice (redundant loop). This evaluator runs as part of the LangSmith evaluate() suite and results appear in the experiment comparison UI.

Intermediate Step Testing:

LangSmith traces each ReAct iteration. We evaluate not just the final response but intermediate steps: (1) After each tool call, was the observation correctly incorporated into the next reasoning step? (2) Did the agent's 'thinking' (visible in trace) correctly identify what information was still needed? (3) When the agent decides to stop, is the reasoning valid? This is implemented as an LLM-as-judge evaluator that scores the full message trajectory, not just the final output. Scored on a 1-5 rubric: reasoning quality, tool efficiency, answer completeness.

ReAct-Specific Mock Strategy:

For Tier 1/2 tests, mock the LLM to return specific ReAct trajectories: (1) **Happy path mock:** LLM returns tool_call for portfolio_summary → mock tool returns data → LLM returns final answer. Tests the full ReAct loop without LLM cost. (2) **Multi-step mock:** LLM returns tool_call #1 → observe → tool_call #2 → observe → final answer. Tests that the graph correctly loops. (3) **Error recovery mock:** LLM calls tool → tool returns error → LLM should retry or gracefully degrade. (4) **Max iteration mock:** LLM keeps calling tools indefinitely → verify the agent halts at max iterations. These mocks use vi.fn().mockResolvedValueOnce() to control the exact sequence of LLM responses.

Decision: ReAct testing = trajectory-based. Evaluate tool selection, reasoning quality, observation handling, and loop termination. LangSmith trajectory evaluator + LLM-as-judge on intermediate steps. Mock LLM for deterministic ReAct loop testing in Vitest.

Summary: ReAct pattern via createReactAgent. Test the full trajectory, not just final output. LangSmith traces every reason/act/observe step. Custom trajectory evaluator scores tool selection. LLM-as-judge on intermediate reasoning. Mock LLM produces deterministic ReAct sequences for unit tests.

Updated Agent Graph Topology (ReAct)

ReAct Loop: User Query → [System Prompt + Context] → **LLM Reason** → {tool_calls?} → yes: **Execute Tools** → **Observe Results** → back to **LLM Reason** | no: **Verify + Disclaim** → Response. Max iterations: 5 (configurable). Verification is a post-ReAct node that wraps the final output before it reaches the user. The ReAct loop handles dynamic tool orchestration; the verification node handles safety guardrails.

Architecture Summary

Key Architectural Insights

1. Agent wraps existing services. Ghostfolio's NestJS services already handle portfolio calculations, market data, transactions. Tools are thin adapters. Inherit all business logic for free.

2. Verification is architectural. Mandatory LangGraph node makes it structurally impossible to bypass fact-checking. Stronger than prompt engineering alone.

3. Model-agnostic from day one. LLM injected via provider interface. Swap Sonnet for GPT-4o or local models via config. LangSmith captures model metadata for comparison.

4. LangSmith unifies observability + evals. One platform for tracing, eval datasets, experiments, and monitoring. Zero-config with LangGraph. Eval results in CI.

5. Documentation is a deliverable, not an afterthought. Six layers ensure every decision is captured, every API is documented, and the sprint journey is traceable.

6. Testing has clear tiers with clear mocking boundaries. Unit tests never hit network. Integration tests use real Ghostfolio + mock LLM. Agent evals use real everything. Each tier has a purpose and a cost trade-off.

Agent Graph Topology

User Query → [Parse + Route] → [Plan Tool Calls] → [Execute Tools] → [Verify Results + Source Attribution] → [Apply Disclaimers + Confidence Score] → Response. Conditionals: confidence <70% → escalation. Clarification needed → ask user. Verification failure → re-plan or partial results.

Changes from v1

Dimension	v1	v2 (this document)
Observability	Langfuse (50K obs/mo free)	LangSmith (5K traces/mo free, native LangGraph)
Evals	Custom runner + Langfuse	LangSmith evaluate() with datasets + experiments
LLM strategy	Sonnet fixed	Model-agnostic via provider interface
Testing	Mentioned, not detailed	3-tier strategy with mock boundaries + local REPL
Documentation	README only	6-layer: TSDoc + Compodoc + ADR + CLAUDE.md + Swagger + DEVLOG
Build priority	Tasks only	Tasks + documentation deliverable per task