FIRST Make two md files **FINAL_ARCHITECT_OVERVIEW.md, AGENT_ACTION_PLAN.md**

File 1: FINAL_ARCHITECT_OVERVIEW.md

Markdown

Final Architect Overview: AI Trading System

1. Introduction: The Restaurant with a Fake Kitchen

This document provides the definitive architectural overview, current state assessment, and strategic direction for the AI Trading System project. It synthesizes analyses from the initial walkthrough, a comprehensive plan, and a final roadmap document.

To understand the project's current state, we use the "Restaurant Analogy":

* The **frontend (`frontend/`)** is the beautiful, customer-facing dining room. It has a well-designed menu (`UI components`), seating (`React architecture`), and waiters (`routing`). Customers can browse, place orders (`trading actions`), and see their order history (`dashboard`). It *appears* functional.
* The **backend (`app/`)** is the kitchen. It houses advanced chefs (`AI/ML services`), a pantry (`database`), and supplier connections (`broker integration`). It is well-equipped but largely idle.

The Core Problem: The dining room is disconnected from the kitchen. Orders placed in the frontend are *not* sent to the backend. Instead, the frontend uses a self-contained "fake kitchen" (`TradingContext.tsx`) that simulates every function using local browser storage (`localStorage`). The real, powerful backend remains unused for most core functions.

---

2. Current State Assessment

Strengths:

1. **Solid Backend Foundation:** Built with FastAPI, featuring a clear structure (`app/core`, `app/models`, `app/services`), robust security (`app/core/security.py`), proper database setup (SQLAlchemy models, Alembic migrations), and a service layer designed for complex logic. The functional API (`app/api/v1/`) is a high-quality base.
2. **Modern Frontend Architecture:** Uses React/Vite with TypeScript, well-organized

components (`shadcn/ui`), context for state management, and protected routing.
3. **Good Supporting Infrastructure:** Includes database migrations (`alembic/`), containerization (`Dockerfile`), and a foundation for testing (`tests/` with `pytest`).

Critical Weaknesses:

1. **Frontend/Backend Disconnect:** The primary issue. Except for authentication, the frontend does not utilize the backend API for its core trading, portfolio, or data display functions.
2. **Frontend Mock Implementation:** The `TradingContext.tsx` acts as a complete fake backend, duplicating logic and preventing real data persistence or multi-user functionality.
3. **Incomplete Backend Services:** While numerous services exist in `app/services/`, many are non-functional placeholders, skeletons lacking implementation, or rely on the unimplemented `ZerodhaService` for essential data. Key features like backtesting and fundamental analysis are currently inoperable.
4. **Significant Code Redundancy:** Multiple instances of duplicate or conflicting code exist (e.g., mock APIs in `app/api/endpoints/`, `sentiment.py` vs. `sentiment_analysis.py`, `report_generator.py` implementations, fragmented frontend utils).
5. **Unclear ML Model:** The pre-trained `ml_model.joblib` is opaque (unknown inputs/outputs) and has a version mismatch, posing a significant integration risk.
6. **Fragmented Testing & CI/CD:** An unimplemented test plan (`testsprite_tests/`) exists alongside the partial `pytest` suite (`tests/`). CI/CD setup is present but conflicting and likely non-functional (`.github/workflows/`).
7. **Ambiguous Deployment Strategy:** Evidence of both Docker/SSH (`Dockerfile`, `ci.yml`) and Vercel (`vercel.json`) deployment targets exists.

Conclusion: The project is a **prototype/proof-of-concept** with well-built but disconnected foundations and significant unimplemented functionality. It requires substantial integration work, backend service implementation, and cleanup to become production-ready.

---

3. Target State Architecture

The goal is a fully integrated, production-ready AI Trading System:

1. **Integrated Application:** Frontend seamlessly communicates with the backend via REST APIs for all data and actions. The mock `TradingContext` is eliminated.
2. **Functional Backend:** All core backend services (Trading, Portfolio, Market Data) and a prioritized set of AI/ML/Financial Analysis services are fully implemented, fetching real data (via broker integration or reliable libraries like `yfinance`). Placeholder logic is replaced.
3. **Connected & Responsive Frontend:** UI accurately reflects real-time backend data, handles loading/error states gracefully, and utilizes `react-query` for server state management.

New UI components are built for implemented backend features.
4. **Robust Quality Assurance:** A unified `pytest` suite provides comprehensive backend test coverage (unit, integration, API). Basic frontend tests validate critical components. Tests are integrated into CI.
5. **Streamlined DevOps:** A single, functional CI/CD pipeline (based on `ci-cd.yml`) handles testing, linting, security scans, building, and deployment using a unified Docker strategy. Vercel deployment is deprecated.
6. **Clean & Maintainable Codebase:** All redundant code, mock APIs, and conflicting implementations are removed. Code follows consistent standards (enforced by linting/formatting in CI).
7. **Clear Documentation:** Consolidated documentation in `/docs` provides a complete guide for developers.

---

4. Strategic Approach & Critical Path

The transformation will follow a phased approach focusing on enabling core functionality first:

1. **Foundation & Cleanup:** Remove dead code and consolidate utilities/configs.
2. **Core Integration (Critical Path):**
   * **Implement Data Source / Broker Integration (`ZerodhaService` or alternative):** This is the **highest priority blocker** as many services depend on it. Implement either live trading via Kite Connect API or use reliable libraries (`yfinance`) for data fetching if live trading is deferred.
   * **Rewrite Frontend `TradingContext`:** The second critical path item, necessary for *any* user-facing functionality. Replace mock logic with backend API calls using `react-query`.
3. **Backend Service Implementation:** Prioritize and implement key AI/ML/Financial Analysis services based on dependencies (data source readiness) and value.
4. **Frontend Expansion:** Build the UI components and pages required for the newly implemented backend services.
5. **Testing & QA:** Consolidate test efforts into `pytest`, implement the `testsprite` plan using `pytest`, and expand coverage.
6. **Production Hardening & DevOps:** Implement security measures, fix/finalize CI/CD, establish Docker deployment, and prepare operational scripts.
7. **Documentation:** Consolidate and update all project documentation.

This structured approach, detailed in the `AGENT_ACTION_PLAN.md`, ensures that foundational issues are addressed before expanding features, leading to a stable and functional application.

File 2: AGENT_ACTION_PLAN.md

Markdown

```
# Agent Action Plan: AI Trading System

This document outlines the specific, ordered tasks required to transform the AI Trading System from its current prototype state into a fully functional, production-ready application. Follow these steps methodically.

**Core Objective:** Connect the disconnected frontend and backend, implement placeholder services, clean up the codebase, establish robust testing and CI/CD, and prepare for deployment.

---

## Phase 1: Foundational Cleanup & Setup (~1-2 days)

**Goal:** Establish a clean, consistent baseline.

* **Task 1.1: Delete Mock Backend APIs**
  * **Action:** Delete the entire `app/api/endpoints/` directory.
  * **Files:** `app/api/endpoints/*`
  * **Guidance:** These are non-functional mocks superseded by `app/api/v1/endpoints/`.
* **Task 1.2: Update Main API Router**
  * **Action:** Edit `app/api/api.py` to remove all import statements and `include_router` calls referencing the deleted mock endpoints in `app/api/endpoints/`. Ensure only routers from `app/api/v1/endpoints/` are included.
  * **Files:** `app/api/api.py`
* **Task 1.3: Consolidate Backend Service Implementations**
  * **Action:** Delete `app/services/sentiment_analysis.py`. The functional service is `app/services/sentiment.py`. Update any imports.
  * **Files:** `app/services/sentiment_analysis.py`
  * **Action:** Resolve conflicting implementations in `app/services/report_generator.py`.
**Decision:** Deprecate `ResearchReport` class and `generate_research_report` function. Refactor `ReportGenerator` class to be the sole implementation, removing dead code.
  * **Files:** `app/services/report_generator.py`
* **Task 1.4: Consolidate Frontend API Utilities & Utils**
  * **Action:** Create a single `frontend/src/lib/api.ts` module.
    * Move `axios` instance setup and interceptors from `frontend/src/lib/api-services.ts`.
```

    * Move API constants and endpoint definitions from `frontend/src/utils/api.ts`.
    * Move all data-fetching functions (e.g., `getStockDetails`) from `frontend/src/lib/api-services.ts` into this new module.
  * **Action:** Move React Query hooks from `frontend/src/hooks/use-api.ts` to a new file `frontend/src/lib/queries.ts`. Update imports.
  * **Action:** Consolidate general utility functions. Delete one of the duplicate files: `frontend/src/utils.ts` or `frontend/src/utils/utils.ts`. Ensure the remaining file (preferably `frontend/src/lib/utils.ts` alongside `api.ts`) contains the necessary functions (`cn`).
  * **Action:** Delete the old, now empty/redundant files: `frontend/src/lib/api-services.ts`, `frontend/src/utils/api.ts`, `frontend/src/hooks/use-api.ts`, and the duplicate utils file.
  * **Action:** Perform a global search/replace in the `frontend/src/` directory to update all import paths to the new consolidated locations (`frontend/src/lib/api`, `frontend/src/lib/queries`, `frontend/src/lib/utils`).
  * **Files:** `frontend/src/lib/api-services.ts`, `frontend/src/utils/api.ts`, `frontend/src/hooks/use-api.ts`, `frontend/src/utils.ts`, `frontend/src/utils/utils.ts`, `frontend/src/**/*` (for imports)
* **Task 1.5: Environment Variable Setup**
  * **Action:** Review both `.env.example` (root) and `frontend/.env.example`. Create functional `.env` and `frontend/.env` files locally.
  * **Action:** Document *all* required environment variables (backend and frontend) in `docs/environment.md`. Specify defaults and whether they are required for core functionality vs. optional features (e.g., `ZERODHA_API_KEY`).
  * **Files:** `.env.example`, `frontend/.env.example`, `docs/environment.md` (new)
* **Task 1.6: Setup Logging & Error Handling Framework**
  * **Action:** Review `app/core/logging.py` and `app/middleware/error_handler.py`. Ensure structured logging (using the configured logger) and consistent API error responses (using FastAPI exception handlers and standard HTTP status codes) are established early.
  * **Files:** `app/core/logging.py`, `app/middleware/error_handler.py`, `app/main.py` (to register handlers)

Phase 2: Critical Path Implementation (Core Functionality) (~2-3 weeks, depends heavily on 2.1)

Goal: Connect the frontend and backend for basic trading and data display. Implement the primary data source.

* **Task 2.1: Implement Data Source / Broker Integration (BLOCKER)**
  * **Priority:** **Highest**. This blocks many other backend services.
  * **Action:** Implement the `app/services/zerodha_service.py`.
  * **Decision Guidance:**
    * **Option A (Live Trading):** If Zerodha API keys **ARE** available, fully implement the Kite Connect API calls for `login`, `get_profile`, `get_holdings`, `get_positions`, `place_order`,

`get_instruments`, `get_quote`, `get_historical_data`, and WebSocket connection for live ticks. Remove all paper trading fallback logic. Use `httpx` for async API calls. Implement robust error handling for API failures.
    * **Option B (Reliable Data - No Live Trading):** If Zerodha keys **ARE NOT** available or live trading is deferred, remove the Kite Connect dependency *for now*. Refactor the service (or create a new `MarketDataService`) to use `yfinance` or another reliable library for *all* required data fetching (`get_quote`, `get_historical_data`, `get_instruments` - potentially from a static list). Ensure consistent data structures are returned. Paper trading logic can remain for simulated order execution but ensure it uses the fetched market data correctly.
  * **Files:** `app/services/zerodha_service.py`, `app/models/zerodha.py` (if using paper trading DB)
* **Task 2.2: Rewrite Frontend `TradingContext` (BLOCKER)**
  * **Priority:** **Highest** (concurrent with 2.1). This blocks all UI functionality.
  * **Action:** Completely rewrite `frontend/src/context/TradingContext.tsx`.
    1. Remove *all* `useState` variables acting as the fake database (`virtualCash`, `transactions`, `watchlist`, `portfolio`).
    2. Remove *all* `useEffect` hooks saving/loading from `localStorage`.
    3. Refactor every function (`buyStock`, `sellStock`, `addToWatchlist`, etc.) to be an `async` function.
    4. Inside these functions, call the corresponding backend API endpoints using the consolidated API service (`frontend/src/lib/api.ts`). Use `useMutation` hooks from `react-query` (`@tanstack/react-query`) for actions that modify data (buy, sell, update watchlist).
    5. Do *not* store portfolio, watchlist, or transaction data directly in this context. Rely entirely on `react-query` to fetch and cache this server state. This context might only be needed to expose the mutation functions.
  * **Files:** `frontend/src/context/TradingContext.tsx`, `frontend/src/lib/api.ts`, `frontend/src/lib/queries.ts`
* **Task 2.3: Refactor Core UI Components**
  * **Action:** Go through pages and components currently using `useTrading()`: `Dashboard.tsx`, `Trading.tsx`, `StockDetails.tsx`, `Transactions.tsx`, `PortfolioSummary.tsx`, etc..
  * **Action:** Remove calls to `useTrading()` for fetching data.
  * **Action:** Replace data fetching with the appropriate `react-query` hooks from `frontend/src/lib/queries.ts` (e.g., `usePortfolio`, `useWatchlist`, `useTransactions`).
  * **Action:** Implement UI states based on `react-query` hook status (`isLoading`, `isError`, `data`). Use `Skeleton` components for loading states and `Alert` components or `Toast` notifications for errors.
  * **Action:** Connect action buttons (Buy, Sell, Add to Watchlist) to the mutation functions exposed by the rewritten `TradingContext` or directly via `useMutation` hooks.
  * **Files:** `frontend/src/pages/*.tsx`, `frontend/src/components/dashboard/*.tsx`, `frontend/src/components/TradingActions.tsx`, etc.

Phase 3: Backend Service Implementation & API Expansion (~2-4 weeks, depending on service complexity)

Goal: Implement placeholder services and expose them via APIs. Prioritize based on dependencies and perceived value.

* **Task 3.1: Implement `backtest.py`**
  * **Action:** Implement the `_get_historical_data` method using the data source established in Task 2.1 (`ZerodhaService` or `yfinance`).
  * **Action:** Implement `analyze_results` helpers to calculate Sharpe ratio, Max Drawdown, CAGR, etc., using standard financial formulas.
  * **Action:** Create API endpoints in `app/api/v1/endpoints/backtests.py` (or similar) for `run_backtest`, `optimize_parameters`, and fetching results. Use Pydantic models for request/response validation. Ensure proper background task handling (e.g., using `FastAPI BackgroundTasks` or Celery) for potentially long-running backtests/optimizations.
  * **Files:** `app/services/backtest.py`, `app/api/v1/endpoints/backtests.py`, `app/schemas/trading.py` (add backtest schemas)
* **Task 3.2: Implement `fundamental_analysis.py`**
  * ***Action:** Implement data fetching (`_get_balance_sheet`, etc.) using the data source from Task 2.1 or `yfinance`'s `Ticker` object methods (`.info`, `.financials`, `.balance_sheet`, etc.).*
  * ***Action:** Implement financial ratio calculations based on standard formulas.*
  * ***Action:** Create an API endpoint in a new file `app/api/v1/endpoints/analysis.py` to expose `get_comprehensive_analysis`. Ensure a well-structured response using Pydantic models. Use caching (`@cache_response`) aggressively here.*
  * ***Files:** `app/services/fundamental_analysis.py`, `app/api/v1/endpoints/analysis.py` (new), `app/schemas/market.py` (add analysis schemas)*
* ***Task 3.3: Implement `report_generator.py` (Consolidated)***
  * ***Action:** Implement the placeholder methods in the refactored `ReportGenerator` class by calling functional services (`sentiment.py`, `competitor.py`, `fundamental_analysis.py` once implemented).*
  * ***Action:** Implement basic PDF generation using `reportlab` based on fetched data. Consider a simple AI summary using an LLM API if available, otherwise use a template.*
  * ***Action:** Create an API endpoint in `app/api/v1/endpoints/reports.py` to trigger report generation (potentially as a background task) and return the report (e.g., as a downloadable file or link).*
  * ***Files:** `app/services/report_generator.py`, `app/api/v1/endpoints/reports.py` (new or refactor existing mock), `app/schemas/report.py` (new or refactor)*
* ***Task 3.4: Address `ml_model.joblib` / `ml_predictions.py`***
  * ***Action:** Follow the solution from the Architect Overview: Attempt introspection (load model, check `feature_names_in_`, `classes_`, log version warning).*
  * ***Action:** If unclear, **implement a simple, transparent placeholder model** (e.g., moving average crossover strategy signal) within `ml_predictions.py`. Clearly document this change.*
  * ***Action:** Ensure the `app/api/v1/endpoints/ml.py` endpoint correctly calls the (potentially new placeholder) prediction logic and uses appropriate Pydantic models. Document the placeholder status clearly in code comments and `docs/`.*
  * ***Files:** `app/services/ml_predictions.py`, `ml_model.joblib` (potentially discard),*

`app/api/v1/endpoints/ml.py`
** **Task 3.5: Enhance `screener.py`***
* * **Action:** Replace the hardcoded stock list with a dynamic list from the Task 2.1 data source (e.g., fetched instruments).*
* * **Action:** Create an API endpoint in `app/api/v1/endpoints/screener.py` accepting filter criteria via Pydantic model and returning matching stocks. Implement pagination using `app/utils/pagination.py`.*
* * **Files:** `app/services/screener.py`, `app/api/v1/endpoints/screener.py` (new or refactor mock), `app/schemas/market.py` (add screener schemas)*
** **Task 3.6: Implement Other High-Value Services (Selectively)***
* * **Action:** Implement APIs and necessary service logic for:*
* * `forecasting.py`: Requires API endpoint and Pydantic models.*
* * `sentiment.py`: Ensure robust API endpoint exists.*
* * `technical_analysis.py`: Implement placeholder patterns/support-resistance if feasible. Requires API endpoint.*
* * `optimizer.py`: Review/replace logic if needed. Requires API endpoint.*
* * `competitor.py`: Fix logic. Requires API endpoint.*
* * Consider implementing `anomaly_detection.py`, `regime_detection.py`, `event_impact.py` if time permits and value is clear.*
* * **Action:** For *each* service implemented, create corresponding API endpoints in `app/api/v1/endpoints/`, ensuring RESTful design and Pydantic validation. Add necessary schemas in `app/schemas/`. Apply caching where appropriate. Implement consistent error handling.*
* * **Files:** `app/services/*.py`, `app/api/v1/endpoints/*.py`, `app/schemas/*.py`*

Phase 4: Frontend Expansion & Refinement (~2-3 weeks)

***Goal:** Build UI for newly implemented backend features and refine existing UI.*

** **Task 4.1: Build UI for Implemented Services***
* * **Action:** Connect existing placeholder pages to their corresponding backend APIs using `react-query` hooks from `frontend/src/lib/queries.ts`:*
* * `Screener.tsx` -> Screener API*
* * `Optimizer.tsx` -> Optimizer API*
* * `Research.tsx` -> Fundamental Analysis API, Competitor API, Sentiment API*
* * `Reports.tsx` -> Report Generator API*
* * **Action:** Design and build *new* pages/components for high-value features implemented in Phase 3:*
* * **Backtesting:** UI for config, running, results (equity curve, metrics table). Needs charting lib.*
* * **Forecasting:** UI for config, displaying forecast chart. Needs charting lib.*
* * **Technical Analysis:** Integrate TA indicators display into `StockChart.tsx`. Needs charting lib integration.*
* * **Action:** Ensure all new UI includes proper loading (`Skeleton`), error (`Alert`/`Toast`), and empty states.*
* * **Files:** `frontend/src/pages/*.tsx`, `frontend/src/components/**/*.tsx`,*

`frontend/src/lib/queries.ts`, `frontend/src/lib/api.ts`
*** **Task 4.2: Frontend State Management Review****
  *** **Action:** Ensure clear separation between server state (`react-query`) and global UI state (React Context/Zustand for theme, auth status). Refactor if necessary.*
  *** **Files:** `frontend/src/context/*.tsx`, `frontend/src/App.tsx`, potentially introduce Zustand store.*
*** **Task 4.3: UI Component Library Assessment & Charting****
  *** **Action:** Evaluate if `shadcn/ui` components are sufficient for new UIs, especially charts.*
  *** **Action:** Add and configure a charting library (e.g., `recharts`, `react-chartjs-2`, `tremor`). Integrate it into `StockChart.tsx` and new components for Backtesting/Forecasting.*
  *** **Files:** `frontend/src/components/ui/`, `frontend/package.json`, `frontend/src/components/dashboard/StockChart.tsx`, new chart components.*

Phase 5: Testing & Quality Assurance (~1-2 weeks)

__Goal:__ Ensure application correctness and stability through comprehensive testing.

*** **Task 5.1: Discard `testsprite_tests`****
  *** **Action:** Delete the entire `testsprite_tests/` directory and related root files (`run_testsprite_tests.py`, `testsprite_config.json`). Its value is informational (test plan) and diagnostic (report).*
  *** **Files:** `testsprite_tests/`, `run_testsprite_tests.py`, `testsprite_config.json`*
*** **Task 5.2: Implement Backend Test Plan in `pytest`****
  *** **Action:** Use `testsprite_backend_test_plan.json` (before deleting) and `tests/TEST_PLAN.md` as requirements.*
  *** **Action:** Implement API integration tests in `tests/api/` for all functional endpoints, covering scenarios from the `testsprite` plan (auth flow, trading operations, etc.). Use the `client` fixture from `conftest.py`. Fix bugs identified in the `testsprite-mcp-test-report.md` (e.g., login username/email mismatch).*
  *** **Action:** Write unit tests for complex logic within services (e.g., financial calculations, backtest metrics, prediction logic if custom model is used). Place these in corresponding test files (e.g., `tests/services/test_fundamental_analysis.py`).*
  *** **Action:** Ensure tests cover success cases, failure cases (invalid input, permissions), and edge cases. Mock external dependencies (like broker APIs or `yfinance`) where necessary for unit tests.*
  *** **Files:** `tests/**/*.py`, `tests/conftest.py`*
*** **Task 5.3: Implement Basic Frontend Tests****
  *** **Action:** Set up Vitest and React Testing Library (confirm config in `vite.config.ts`).*
  *** **Action:** Write basic rendering and interaction tests for critical components: `Login.tsx`, `TradingActions.tsx`, key forms for Screener/Optimizer/Backtesting. Focus on ensuring elements render and form submissions can be initiated.*
  *** **Files:** `frontend/src/**/*.test.tsx` (new), `frontend/vite.config.ts`*

Phase 6: Production Hardening & DevOps (~1 week)

***Goal:** Secure the application, finalize CI/CD, and establish a deployment strategy.*

** **Task 6.1: Implement Security Measures***
* * **Action:** Modify `Dockerfile` to create and run the application as a non-root user.*
* * **Action:** Implement CSRF protection (e.g., using `python-multipart` and FastAPI's `Request` object with form data or custom middleware checking headers/tokens) in `app/core/security.py` or `app/main.py`.*
* * **Action:** Refactor `RateLimitMiddleware` in `app/core/security.py` to use Redis (via `app.core.cache`) for distributed rate limiting.*
* * **Action:** Implement JWT token blacklisting on logout (e.g., storing revoked JTI in Redis with expiry) within the authentication logic in `app/api/v1/endpoints/auth.py` and `app/core/security.py`.*
* * **Files:** `Dockerfile`, `app/core/security.py`, `app/main.py`, `app/api/v1/endpoints/auth.py`, `app.core.cache.py`*
** **Task 6.2: Finalize CI/CD Pipeline***
* * **Action:** Consolidate GitHub Actions workflows. **Decision:** Use `ci-cd.yml` as the base.*
* * **Action:** Merge essential steps from `test.yml` (multi-python testing, `bandit`, `safety`) and `ci.yml` (linting/formatting checks - `black`, `flake8`, `mypy`) into the `test` job of `ci-cd.yml`. Configure tools via `pyproject.toml` where possible.*
* * **Action:** Ensure the `test` job runs the complete `pytest` suite (from Phase 5) using `requirements-test.txt`.*
* * **Action:** Configure the `build` job to build the final multi-stage Docker image (from Task 6.3) and push it to a container registry (e.g., Docker Hub, GHCR).*
* * **Action:** Configure the `deploy` job for the chosen strategy (e.g., SSH into server and run `docker-compose pull && docker-compose up -d`, or trigger a Kubernetes deployment update). Use GitHub secrets for credentials/keys.*
* * **Action:** Delete `ci.yml` and `test.yml`.*
* * **Files:** `.github/workflows/ci-cd.yml`, `.github/workflows/ci.yml` (delete), `.github/workflows/test.yml` (delete), `pyproject.toml`*
** **Task 6.3: Define and Implement Deployment Strategy***
* * **Action:** **Decision:** Standardize on Docker deployment. Deprecate Vercel strategy.*
* * **Action:** Create/Refine a multi-stage `Dockerfile`. Stage 1 (`node` base): install frontend deps, build static assets. Stage 2 (`python` base): install backend deps (`requirements.txt`). Final stage (`python-slim` base): copy backend code from Stage 2, copy built frontend assets from Stage 1 into a `./static` directory, install runtime deps only, create non-root user, run `uvicorn` via `CMD`. Ensure FastAPI serves static files from `./static`.*
* * **Action:** Create `docker-compose.yml` defining services for the backend application, PostgreSQL database, and Redis. Use volumes for persistent data. Configure networking and environment variables.*
* * **Action:** Update `ci-cd.yml`'s deploy job to use `docker-compose` commands via SSH.*
* * **Action:** Delete `vercel.json`.*
* * **Files:** `Dockerfile`, `docker-compose.yml` (new), `.github/workflows/ci-cd.yml`, `vercel.json` (delete), `app/main.py` (add static files mount)*
** **Task 6.4: Developer Setup & Seeding Scripts***
* * **Action:** Ensure `scripts/create_tables.py` is **removed** or updated to use Alembic (`alembic upgrade head`). Ensure `alembic/env.py` correctly reads the database URL from*

config. The primary mechanism for schema management must be Alembic.
  *   **Action:** Review and enhance `scripts/seed_database.py` to provide sufficient and realistic sample data (users, strategies, market data if using paper trading mode) for local development and testing. Make it idempotent if possible.*
  *   **Files:** `scripts/create_tables.py` (deprecate/remove), `alembic/env.py`, `alembic.ini`, `scripts/seed_database.py`*

Phase 7: Final Review & Documentation (~1-2 days)

__Goal:__ Ensure all documentation is consolidated, accurate, and reflects the final state.

** __Task 7.1: Consolidate Documentation__*
  *   **Action:** Move this `AGENT_ACTION_PLAN.md` to `docs/action-plan.md`.*
  *   **Action:** Move the `FINAL_ARCHITECT_OVERVIEW.md` to `docs/architect-overview.md`.*
  *   **Action:** Merge relevant content from `architecture.md` into `docs/architect-overview.md`. Delete `architecture.md`.*
  *   **Action:** Review and update `docs/deployment.md`, `docs/api_examples.md`, `docs/backup-recovery.md`, ensuring they reflect the final implementation and Docker deployment strategy. Add `docs/environment.md` (from Task 1.5).*
  *   **Action:** Move all other relevant root markdown files (`PROJECT_SUMMARY.md`, `TESTSPRITE_GUIDE.md` - archive if obsolete) into `/docs` or an `/archive` subfolder.*
  *   **Action:** Update the root `README.md` to be concise, explain the project briefly, mention setup using `docker-compose`, and link prominently to `/docs/architect-overview.md` as the main entry point for developers.*
  *   **Files:** All `*.md` files, `/docs/` directory.*

** __Task 7.2: Final Code Review and Cleanup__*
  *   **Action:** Perform a final pass through the codebase, removing any remaining commented-out code, TODOs (that were addressed), or unused imports/variables. Ensure consistent formatting (`black`). Run linters (`flake8`, `mypy`) and fix violations.*
  *   **Files:** Entire codebase (`app/`, `frontend/src/`)*
** __Task 7.3: Pre-Production Check__*
  *   **Action:** Build the final Docker image. Deploy to a staging environment using `docker-compose.yml`.*
  *   **Action:** Run Alembic migration (`alembic upgrade head`) and seeding script in staging.*
  *   **Action:** Conduct end-to-end testing of core user flows in the staging environment.*
  *   **Action:** Verify all environment variables are correctly set for production deployment (using secrets management, not hardcoding).*