Technical Specification: SignalForge AI Stock Analyst

This document provides a detailed overview of the system architecture, component design, and end-to-end data flows for the SignalForge AI Stock Analyst platform.

1. High-Level Design (HLD)

The platform is built on a microservices-oriented architecture, coordinated by an API Gateway and orchestrated via Docker Compose.

System Architecture Diagram

```
graph TD
    User((User/Browser)) -->|HTTPS/WSS| Gateway[API Gateway - FastAPI]

    subgraph "Backend Services"
        Gateway -->|Proxy| Ingestion[Ingestion Service - FastAPI]
        Gateway -->|Proxy| RecEngine[Recommendation Engine - FastAPI]
        Gateway -->|Proxy| Predict[Prediction Service - XGBoost/FastAPI]
        Gateway -->|Proxy| Auth[Auth Service - Firebase/GSI]
        Gateway -->|Proxy| Trading[Trading Service - Paper/Live]
    end

    subgraph "Data Ingestion (Scrapers/APIs)"
        Ingestion -->|Web Scraping| LiveMint[LiveMint/Moneycontrol]
        Ingestion -->|API| YF[yfinance / StockData]
        Ingestion -->|Crawl| Screener[Screener.in / TickerTape]
        Ingestion -->|Social| Reddit[Reddit / ValuePickr]
    end

    RecEngine -->|Request Prediction| Predict
    RecEngine -->|Read Crawled Data| Data[(Local JSON Storage / DB)]
    Ingestion -->|Write Crawled Data| Data
```

Component Breakdown

- **API Gateway (Port 8000)**: Central entry point handling CORS, rate-limiting, and routing to internal services.
- **Ingestion Service (Port 8002)**: Orchestrates distributed crawlers for fundamental and technical data.
- **Recommendation Engine (Port 8001)**: The brain of the system, fusing multiple signal layers using the `ScoringModel`.
- **Prediction Service (Port 8003)**: Hosts the XGBoost ML model for quantitative probability scoring.
- **Frontend (Port 3000)**: React-based dashboard with real-time updates and interactive data visualization.

---

2. Low-Level Design (LLD)

Scoring Engine Logic (S.T.A.F.A)

The `ScoringModel` follows a multi-layer fusion strategy to calculate a final "Conviction" score (0-100%).

Layer	Component	Weight (Base)	Logic Description
S	**Sentiment**	20%	Aggregates Reddit, ValuePickr, and News polarity (Freshness-weighted).
T	**Technical**	30%	Continuous RSI mapping, MA trends, and Volatility (ATR) ratios.
A	**AI Model**	20%	XGBoost probability refined by an "AI Confidence" metric.
F	**Fundamental**	15%	Sector PE comparison, ROCE/ROE thresholds, and Pros/Cons parsing.
A	**Analyst**	15%	TickerTape upside percentage and professional BUY/SELL consensus.

Regime Detection & Dynamic Gating

The system detects the market state to shift weights dynamically:

- **TRENDING**: Boosts Technical weights.
- **CHOP**: Boosts ML/XGBoost weights.
- **VOLATILE**: Triggers risk penalties and favors Fundamental/Analyst signals over short-term technicals.

News & Ticker Logic

The `TickerBar` component implements a hybrid display strategy:

- **yfinance News**: Rendered as clickable links for deep-dives into general market news.
- **LiveMint News**: Rendered as non-clickable headlines for real-time market awareness without external redirects.

---

3. Core Flows & Sequences

End-to-End Analysis Flow

1. **Trigger**: User clicks "Scan Now" or `pipeline_runner.py` is executed via CLI.
2. **Ingestion**:
   - `GlobalMarketCrawler` fetches Nifty 50, VIX, and Global Index status.
   - Specialized crawlers (Screener, Moneycontrol) fetch symbol-specific metrics.
3. **Recommendation**:
   - `RecEngine` receives signals and metadata.
   - `ScoringModel` calculates conviction, identifies regime, and calculates SL/Target levels based on current volatility.
4. **Persistence**: The final recommendation is saved to `recommendations.json` with a unique ID and "active" status.
5. **UI Update**: Frontend fetches the update via REST or updates live via WebSocket.

Rationale Generation Flow

The `RationaleRenderer` on the frontend parses the `rationale` string, which is generated using Natural Language templates in the backend. It applies dynamic styling:

- **Keywords** like "Oversold" or "Bullish" are highlighted using theme-aware colors.
- **Color-coded badges** are generated for each S.T.A.F.A layer's performance.

---

4. Environment Configuration

The system uses a centralized `shared/config.py` coupled with `.env` management to handle:

- Service URLs and Ports.
- Weight Parameters for the Scoring Engine.
- Credential Management for external APIs.
- Logging levels and persistence paths.

---

5. Deployment Strategy

The platform supports a dual-deployment strategy for local development and scalable production environments.

Local Development (Bash-based)

For quick iterations without Docker overhead:

- **Service Management**: Use `manage_services.sh [start|stop|restart]`.
- **Hot-Reloading**: Python services use `uvicorn --reload`; Frontend uses `vite`.
- **Logging**: Centrally stored in the `/logs` directory.

Containerized Deployment (Docker)

The primary strategy for professional deployment and CI/CD:

Component	Strategy	Base Image
Backend Services	Individual Dockerfiles per service.	`python:3.9-slim`
Frontend	Static build served via Nginx (Production).	`node:18-alpine`
Orchestration	`docker-compose.yml` for multi-container coordination.	N/A

Production Pipeline (Recommended)

1. **CI State**: Github Actions / GitLab CI runs linting and unit tests.
2. **Build State**: Multi-stage Docker builds to keep image sizes small.
3. **Environment**: Managed via `.env` files mounted as secrets in Docker.
4. **Monitoring**:
    - **Health Check**: Each service implements a `/health` endpoint.
    - **Persistence**: Data stored in `data/` should be mounted as a persistent volume.
    - **Scaling**: Services are stateless, allowing horizontal scaling behind a Load Balancer (or the API Gateway).