ChatGPT

Blueprint for Offline-First Market Monitor App

Root directory structure

Below is a suggested repository layout for `market_app`. Each folder is accompanied by a brief description. The skeleton layout should be created for an empty first commit.

```
market_app/
│  README.md                - high-level overview and install instructions
│  requirements.txt         - pinned Python dependencies
│  pyproject.toml           - packaging metadata (optional)
│  LICENSE
│  .gitignore
├─ src/                      - core Python packages
│  ├─ market_app/
│  │   ├─ __init__.py       - package init
│  │   ├─ cli.py            - CLI entry point (module A)
│  │   ├─ config.py         - config loading/validation utilities
│  │   ├─ ingestion/
│  │   │   ├─ symbols.py    - read and normalize symbol lists
│  │   │   ├─ ohlcv.py      - read OHLCV caches
│  │   │   ├─ etf.py        - optional ETF holdings ingestion
│  │   │   ├─ edgar.py      - optional EDGAR ingestion
│  │   │   ├─ provenance.py - track data provenance & licensing
│  │   ├─ corpus/
│  │   │   ├─ loader.py     - text corpus ingestion and chunking
│  │   │   ├─ embeddings.py - embedding model wrapper and vector index build
│  │   │   ├─ retrieval.py  - hybrid (BM25 + vector) retrieval
│  │   ├─ features/
│  │   │   ├─ price_features.py  - returns and moving averages
│  │   │   ├─ risk_features.py   - volatility, drawdown, downside risk
│  │   │   ├─ liquidity.py      - ADV and zero-volume metrics
│  │   │   ├─ quality.py        - missingness and split checks
│  │   ├─ regime/
│  │   │   ├─ macro.py      - macro series ingestion and z-score regimes
│  │   │   ├─ scenario.py   - scenario sensitivity definitions
│  │   ├─ themes/
│  │   │   ├─ classifier.py - ETF/EDGAR/sector based theme classification
│  │   │   ├─ taxonomy.py   - theme definitions and mapping
│  │   ├─ scoring/
│  │   │   ├─ gates.py      - eligibility gating rules
```

```
│   │   │   ├── score.py       – score computation (core vs regime overlay)
│   │   │   ├── flags.py       – risk flag taxonomy
│   │   ├── reporting/
│   │   │   ├── report.py      – Markdown report generator
│   │   │   ├── explain.py     – explanation packs and analog evidence
│   │   ├── storage/
│   │   │   ├── io.py          – CSV/Parquet/DuckDB I/O
│   │   │   ├── manifest.py    – manifest writer with git/config hashes
│   │   └── utils/
│   │       ├── logging.py     – centralized logging config loader
│   │       ├── dates.py       – trading-calendar utilities
│   │       ├── caching.py     – offline-mode file checks and download fallback
│   └── tests/                 – pytest unit and smoke tests
├── scripts/
│   ├── run.ps1              – PowerShell script to execute a full run
│   └── provision_data.ps1   – one-time data/model provisioning script
├── config/
│   ├── config.yaml          – main app configuration
│   ├── sources.yaml         – dataset locations and refresh cadence
│   ├── watchlists.yaml      – theme buckets and allowed universe
│   └── logging.yaml         – Python logging configuration
├── data/
│   ├── raw/                 – immutable raw data
│   │   ├── symbols/         – symbol directories from exchanges
│   │   ├── ohlcv/           – per-symbol OHLCV CSVs
│   │   ├── macro/           – macro time-series CSVs
│   │   ├── etf_holdings/    – optional ETF holdings caches
│   │   ├── edgar/           – optional EDGAR filing caches
│   │   ├── metadata/        – data provenance JSONs
│   │   └── text_corpus/raw/  – historical/geopolitical documents
│   ├── interim/             – intermediate processed data
│   ├── processed/           – cleaned and enriched data
│   ├── cache/               – offline caches (e.g., pre-computed features)
│   ├── index/               – FAISS/BM25 indexes and mappings
│   ├── duckdb/              – optional DuckDB database (market.duckdb)
│   └── text_corpus/processed/ – tokenized/corpus chunks with embeddings
├── models/
│   ├── embeddings/          – downloaded embedding models (e.g., SBERT)
│   ├── regime/              – saved clustering models and calibrations
│   ├── scoring/             – optional ML ranking models
│   └── version.txt          – version metadata
├── outputs/
│   └── runs/<timestamp>/    – run-specific artifacts (CSV, JSON, MD, manifest)
│       ├── universe.csv
│       ├── classified.csv
│       ├── features.csv
```

```
│       ├─ eligible.csv
│       ├─ scored.csv
│       ├─ regime.json
│       ├─ report.md
│       └─ manifest.json
└─ docs/
    ├─ architecture.md      – design overview and module interactions
    ├─ data_provenance.md   – description of data sources & licensing
    └─ usage.md             – how to run and interpret outputs
```

Minimal empty scaffold (first commit)

The following files should exist even before implementation: `README.md`, `requirements.txt`, `src/market_app/__init__.py`, `scripts/run.ps1`, `config/config.yaml`, `config/sources.yaml`, `config/watchlists.yaml`, `config/logging.yaml`, `docs/architecture.md`, `docs/data_provenance.md`, and `.gitignore`.

Module-by-module technical specification

A) CLI and settings

Inputs

- `config.yaml` – main configuration. Must specify gating thresholds, scoring weights, offline mode flag and file paths.
- Environment variables (optional) – used to override sensitive values or paths.

Outputs

- Orchestrates the end-to-end pipeline, writing outputs into `outputs/runs/<timestamp>/` and logs into `logs/` (defined in logging configuration).

Key functions/classes

Function/Class	Description
`parse_args(args: List[str]) -> Namespace`	Parses command-line options: `--config`, `--run_id` (optional), `--offline` (boolean), `--top_n`, etc.
`load_config(path: str) -> Config`	Reads YAML, merges with defaults and environment overrides, validates schema using pydantic.

Function/Class	Description
`cli_main()`	Entry point called by `python -m market_app.cli`. It loads config, initializes logging (via `logging.yaml`), computes a deterministic `run_id`, initializes offline mode (disabling network), and invokes the orchestrator to run ingestion → feature engineering → scoring → reporting.
`offline_guard()`	Utility that checks `config.offline` and raises errors if a code path tries to make a network call.

Failure modes and error handling

- **Missing configuration**: If the config file is not found or invalid YAML, raise `SystemExit` with a clear message.
- **Invalid schema**: Use pydantic to validate required fields; report missing keys and type errors.
- **Offline violations**: wrap network-dependent functions in a decorator that raises an exception if offline mode is enabled.

Testing

- Unit tests should assert that `parse_args` correctly parses flags and that `load_config` merges defaults and environment overrides.
- Smoke test: simulate a run with a tiny dataset; ensure that `cli_main` returns without exception and writes output files.

Logging

- The CLI must initialize logging using `logging.yaml`. Each module obtains a logger via `logging.getLogger(__name__)`. Logs should include timestamps and severity. Write logs both to console and to a file under `outputs/runs/<timestamp>/run.log`.

B) Ingestion module

Inputs

- **Symbols**: Directory containing exchange-provided symbol lists (e.g., NASDAQ `nasdaqlisted.txt`, NYSE `.csv`). Required columns: `Symbol`, `Security Name`, `Market Category`, `Test Issue`, `Financial Status`, `Round Lot Size`.
- **OHLCV**: For each symbol, a local CSV with columns `date`, `open`, `high`, `low`, `close`, `adj_close` (optional), and `volume`. File names should follow `data/raw/ohlcv/<symbol>.csv`.
- **ETF holdings (optional)**: CSVs listing ETF constituents with columns `etf_ticker`, `holding_ticker`, `weight`, `date`.
- **EDGAR filings (optional)**: Local folder with downloaded filings (10-K, 10-Q) in raw text or HTML.

Outputs

- **Universe DataFrame** containing unique tickers and metadata (exchange, sector, listing date). Saved as `universe.csv` in the run folder.
- **OHLCV DataFrame** per symbol loaded into memory or DuckDB for feature calculation.
- **Holdings DataFrame**: aggregated by symbol for theme classification.
- **Filing texts**: passed to theme classifier and analog engine.

Key functions/classes

Component	Description
`load_symbols(dir_path) -> pd.DataFrame`	Reads multiple exchange files, normalizes column names, drops test issues and ETFs if configured. Adds `exchange` column and ensures uppercase tickers.
`load_ohlcv(symbols: List[str], ohlcv_dir) -> Dict[str, pd.DataFrame]`	Loads each symbol's price history from CSV; checks that the last date matches or exceeds the current date minus tolerance. Handles missing columns by raising `ValueError`.
`load_etf_holdings(dir_path) -> pd.DataFrame`	Aggregates ETF holdings across dates, filters by configured ETF tickers and normalizes tickers.
`load_edgar(dir_path) -> Dict[str, str]`	Reads pre-downloaded EDGAR filing texts; returns a mapping from ticker to concatenated filing text.
`save_provenance(metadata)`	Writes a JSON entry in `data/raw/metadata/` capturing provider, retrieval date, license URL, and checksums for each file.

Failure modes

- Missing or corrupt files: raise an exception that the orchestrator catches and logs. In offline mode, there is no fallback; the user must provision data before running.
- Staleness: if OHLCV data is older than `config.max_lag_days`, emit a warning flag and still process but mark the symbol as stale.

Testing

- Unit tests for each loader, verifying that missing columns raise errors and that duplicates are removed.
- Integration test that loads sample OHLCV files and asserts the resulting DataFrame shapes.

Logging

- Log the number of symbols loaded and the date range of OHLCV data. Log warnings when files are missing or stale.

C) Text corpus ingestion and indexing

Inputs

- `data/text_corpus/raw/` containing plain text or PDFs of historical and geopolitical documents. All documents must be licensed for offline use.
- `models/embeddings/` containing a sentence-transformer model downloaded during provisioning.

Outputs

- **Chunks DataFrame**: Each row corresponds to a chunk with fields `chunk_id`, `doc_id`, `text`, `tokens`, and optional metadata (document title, date). Stored in Parquet under `data/text_corpus/processed/chunks.parquet`.
- **FAISS index**: built with dimension equal to embedding size. Saved in `data/index/faiss.index` and accompanied by a `doc_mapping.json` that maps `chunk_id` to metadata.
- **Optional BM25 index**: a pickled BM25 object (e.g., from `rank_bm25`) stored in `data/index/bm25.pkl`.

Key functions/classes

Component	Description
`iter_documents(raw_dir)`	Yields `(doc_id, text, metadata)` from plain text or PDF using `pdfminer.six`. Skips binary files.
`chunk_text(text, max_tokens, overlap_tokens)`	Splits text into overlapping chunks to feed into the embedding model. For example, `max_tokens=512` and `overlap=50`.
`load_embedding_model(path)`	Loads a pre-downloaded sentence-transformer (e.g., `all-MiniLM-L6-v2`).
`build_faiss_index(embeddings, ids)`	Builds a FAISS index (e.g., `IndexFlatIP` for inner product) and writes it to disk.
`build_bm25_index(corpus_tokens)`	Builds a BM25 index over tokenized documents for lexical search.
`search_hybrid(query_text, k)`	Generates a query embedding, performs K-NN search on the FAISS index, performs BM25 search, then fuses results via reciprocal rank fusion (RRF). Returns top-`k` chunk IDs and similarity scores.

Failure modes

- Missing embedding model: raise `FileNotFoundError` prompting the user to run the provisioning script.
- FAISS index corrupt: rebuild index from saved embeddings.

Testing

- Unit tests for tokenization and chunking boundaries.
- Test retrieval returns at least `k` results and that retrieval is deterministic given a fixed seed.

Logging

- Log the number of documents processed, number of chunks created, and time taken to build the index. During search, log the query and the number of hits returned.

D) Feature engineering

The module computes price, risk and liquidity features for each symbol. The computations are purely offline.

Inputs

- OHLCV DataFrames for each symbol (from ingestion).
- Configuration thresholds for lookback periods (1/3/6/12 months), volatility windows (20/60 days) and other parameters.

Outputs

- `features.csv` – cross-sectional table with columns: `symbol`, `as_of_date`, `return_1m`, `return_3m`, `return_6m`, `return_12m`, `sma20_ratio`, `sma50_ratio`, `sma200_ratio`, `pct_days_above_sma200`, `volatility20`, `volatility60`, `downside_vol20`, `worst_5d_return`, `max_drawdown_6m`, `adv20_dollar`, `zero_volume_fraction`, plus quality flags.

Key computations

Feature	Calculation
Returns	For a window of `N` trading days (1M ~ 21 days, 3M ~ 63, 6M ~ 126, 12M ~ 252), compute `(last_close / close_N_days_ago) - 1`.
SMA ratios	Compute simple moving averages of closing price over 20, 50 and 200 days. Ratio is `last_close / smaN`. A price above a moving average is often considered bullish; Investopedia notes that a stock remaining above its 200-day SMA is generally considered in an uptrend [1].
% days above SMA200	Fraction of days in the last 6 months where `close > sma200`. A high fraction suggests sustained momentum.
Volatility20/60	Compute daily log returns, take the standard deviation over a 20- or 60-day rolling window, multiply by `√252` to annualize. Investopedia explains that to annualize daily volatility we multiply by the square root of the number of trading days [2].

Feature	Calculation
Downside volatility	Use downside deviation: include only negative daily returns in the standard deviation calculation and annualize similarly. Downside deviation focuses on negative returns to isolate "bad" volatility [3].
Worst-5D return	Over the last 6 months, compute the minimum 5-day cumulative return (rolling window). Signals vulnerability to short-term crashes.
Max drawdown (6M)	Compute the maximum peak-to-trough decline over the last 126 trading days; defined as the largest percentage drop from a historical peak to a subsequent trough [4].
ADV20 dollar	Average daily volume (units) over the last 20 days times the average close; ADV gauges liquidity. Northstar Risk explains that average daily volume measures liquidity by averaging the number of units traded per day over a chosen lookback window [5].
Zero-volume fraction	Fraction of days in the last 60 days with zero volume; used to flag illiquid microcaps.
Quality checks	Identify missing days, stale price data or suspicious splits (large price jumps without volume). Generate flags (`missing_data`, `stale_data`, `split_suspect`).

Normalization

After calculating raw features, apply cross-sectional normalization: z-score each feature across the universe, optionally winsorizing extremes (e.g., clamp to ±3 standard deviations). Normalized features are stored alongside raw values for scoring.

Testing

- Unit tests verifying correct return and volatility calculations on synthetic data.
- Test that a known constant price series yields zero volatility and drawdown.
- Smoke test: compute features for a small set of symbols and ensure no NaNs.

Logging

- Log per-symbol warnings if there are fewer than `min_history_days` of price data. Log summary statistics (mean/median) for each feature.

E) Regime engine (macro context)

Inputs

- Macro series from `data/raw/macro/` (CSV) such as 10-year minus 2-year Treasury yield (term spread), CPI year-over-year inflation, Federal funds rate, unemployment rate, industrial production,

CBOE VIX, commodity indices, geopolitical risk index (optional). Each CSV must have columns `date` and `value`.
- Configuration thresholds for z-score states.

Outputs

- `regime.json` summarizing current macro conditions and scenario probabilities.
- Intermediate DataFrame with z-scores and state labels for each macro series.

Key functions/classes

Component	Description
`load_macro_series(paths) -> pd.DataFrame`	Loads each macro series and aligns on business dates. Performs forward-fill for missing values.
`compute_zscore(series, lookback_years) -> pd.Series`	For each date, compute `(value - mean) / std` over a trailing window (e.g., 5 years).
`label_regime(z, low_thresh, high_thresh) -> str`	Labels each point as `low`, `neutral` or `high` depending on z-score thresholds. For example, a negative term spread (yield curve inversion) may signal recession risk.
`determine_regime(macro_df) -> dict`	Combines individual series states into an overall regime classification, e.g., “Expansion”, “Contraction”, “Stagflation”. This can be rule-based (if inflation high & growth low then stagflation) or learned via clustering (k-means on z-scores).
`apply_regime_adjustments(scores, regime_state)`	Adjusts scoring weights based on the current regime (e.g., overweight defensive sectors in contraction).

Failure modes

- Missing macro series: log a warning and skip regime adjustments.
- Extreme z-scores due to insufficient history: apply clipping.

Testing

- Unit tests for z-score calculation and regime labeling.
- Integration test verifying that regime adjustments modify scores as expected.

Logging

- Log the latest values and z-scores of each macro series and the resulting regime label.

F) Theme classification

Inputs

- Universe of symbols and sectors from ingestion.
- ETF holdings (optional) mapping ETFs to their constituent tickers and weightings.
- EDGAR filings text (optional) for keyword evidence.
- `watchlists.yaml` defining theme buckets, seed tickers, and allowed sectors.

Outputs

- `classified.csv` with columns: `symbol`, `theme`, `confidence`, `evidence` (JSON string with details).

Key functions/classes

Component	Description
`load_theme_definitions(config_path) -> dict`	Reads YAML to obtain theme names (e.g., “Defense”, “StrategicTech”, “CriticalMaterials”), seed tickers and keyword lists.
`classify_by_etf(symbol, etf_holdings)`	Checks whether the symbol is contained in any theme ETF; derives a score based on holding weight and the ETF’s relevance.
`classify_by_keywords(symbol, filings_text, keywords)`	Counts occurrences of theme keywords in filings (10-K/10-Q). Applies TF-IDF weighting and normalizes scores.
`classify_by_sector(symbol, sector_mapping)`	Maps sectors (e.g., GICS) to broader themes.
`aggregate_theme_scores(scores_dict) -> (theme, confidence, evidence)`	Combines scores from different signals using weights specified in `config.theme_weights`. Returns the highest scoring theme, a confidence value (0–1), and evidence (which signals contributed).

Failure modes

- No evidence found: classify as `Unclassified` with zero confidence.
- Multiple themes with equal score: choose the first by alphabetical order and record tie in evidence.

Testing

- Unit tests that classification returns expected themes for known tickers (e.g., `LMT` should map to defense if configured). Provide mocks for ETF holdings and filings.

Logging

- Log the number of symbols assigned to each theme and examples of evidence for debugging.

G) Scoring and flags

Inputs

- Normalized features (from module D).
- Theme classifications (module F).
- Regime state (module E).
- Scoring configuration specifying weights for returns, momentum, volatility, drawdown, liquidity, and theme confidence. Two variants can be defined (conservative vs opportunistic).

Outputs

- `eligible.csv` listing symbols that pass hard gates with columns `symbol` and reasons for inclusion/exclusion.
- `scored.csv` – table containing `symbol`, `score_core`, `score_regime_adjusted`, `total_score`, `flags` (list of strings), and raw/normalized features.

Key functions/classes

Component	Description
`apply_gates(features_df, gates_config) -> (eligible_df, excluded_df)`	Applies hard filters: e.g., `adv20_dollar` ≥ threshold, `zero_volume_fraction` ≤ max, price > SMA200, market cap > config.min_market_cap, etc. Stores reasons for exclusion.
`compute_base_score(normalized_features, weights) -> float`	Weighted sum of normalized returns, momentum and liquidity metrics minus penalties for volatility, downside risk and drawdown.
`apply_regime_overlay(base_score, regime_state, overlay_config) -> float`	Adjusts base score using regime-specific multipliers (e.g., increase weight on quality and liquidity during contractions).
`adjust_for_theme(base_score, theme_confidence, theme_weight)`	Applies a bonus if the symbol belongs to a high-priority theme; penalizes if theme confidence is low or the theme is currently restricted.
`assign_flags(features_df, scoring_df) -> List[str]`	Flags include `HighVolatility`, `LargeDrawdown`, `Illiquid`, `StaleData`, `UnknownMicrocap` etc. Use thresholds in config.

Failure modes

- Division by zero in normalization if all symbols have the same value; handle by adding a small epsilon.
- Negative weights producing unintuitive scores; validate that weights sum to 1 or other constraints.

Testing

- Unit tests for gating: feed sample data to ensure correct inclusion/exclusion.
- Tests for scoring: confirm that higher returns increase scores and high volatility decreases scores.

Logging

- Log how many symbols pass each gate and statistics of scores. Log any negative or NaN scores.

H) Reporting and explainability

Inputs

- Scored DataFrame with scores and flags.
- Eligible universe list.
- Regime summary and scenario sensitivity data.
- Analog retrieval results from module C.
- Config details for report generation (top N, sections to include).

Outputs

- `report.md` – Markdown document summarizing the run, stored in the run folder.
- **Explanation packs** for each top-ranked symbol: JSON files containing raw features, normalized features, scores, flags, theme classification, regime state, and top-`k` analog document snippets with metadata and similarity scores.

Report template

A typical report should include:

1. **Run overview** – timestamp, config hash, dataset range, number of symbols processed and eligible.
2. **Macro/regime summary** – table of current macro indicators with z-scores and regime labels.
3. **Distribution of risk flags** – counts or bar chart (optionally saved as image) showing red/amber/green flag counts.
4. **Top candidates** – a table listing the highest-scoring symbols with key metrics: returns, vol, drawdown, liquidity, theme, score, and flags. Include only short phrases or numbers in the table.
5. **Scenario sensitivity** – narrative explaining how current regime and scenario weights influenced scores (e.g., “The contraction regime penalised high beta names and boosted liquidity-quality names”).
6. **Historical analog evidence** – for the top `k` symbols, list the top `n` retrieved corpus chunks: provide document titles, publication dates and two-sentence snippets with similarity scores. The analog engine uses hybrid search to blend lexical precision and semantic meaning; Elastic’s guide

notes that hybrid search combines lexical BM25 with semantic vector search to improve relevance [6].

7. **Next steps / caveats** – highlight data quality issues, limitations (e.g., survivorship bias not fully eliminated), and suggestions for further analysis.

Failure modes

- Missing analog retrieval results: if the corpus index is unavailable, report gracefully and skip analog evidence.
- Report file I/O errors: catch exceptions and log them; still produce other outputs.

Testing

- Unit tests verifying that Markdown contains the required sections and that tables have the expected number of rows.
- Validate that explanation packs can be parsed as JSON and contain all expected keys.

Logging

- Log the time taken to generate the report and number of explanation packs written.

I) Storage layer

Inputs

- DataFrames and objects produced by other modules.
- Configuration specifying output formats (CSV, Parquet) and optional database use.

Outputs

- Persisted CSV/Parquet files in `outputs/runs/<timestamp>/`.
- Optional DuckDB database `market.db` storing tables such as `ohlcv`, `features`, `classified`, `scored`, etc.
- `manifest.json` capturing run metadata.

Key functions/classes

Component	Description
`write_csv(df, path)`	Writes DataFrame to CSV with UTF-8 encoding and `float_format='%.6f'`.
`write_parquet(df, path)`	Writes DataFrame with compression (snappy or zstd).
`store_duckdb(df_dict, db_path)`	Uses DuckDB Python API to write DataFrames into a database. Creates indices (e.g., on `symbol`, `date`) for faster queries.

Component	Description
`compute_file_checksum(path) -> str`	Computes SHA-256 hash of any file for provenance.
`write_manifest(run_dir, config, git_sha, start_time, end_time)`	Writes a JSON capturing config hash, git commit, run duration, offline flag, versions of dependencies, dataset checksums and model versions.

Failure modes

- File write errors due to permissions or disk space: catch and propagate with meaningful error messages.
- DuckDB exceptions: log and fall back to file-based storage.

Testing

- Unit tests verifying that DataFrames roundtrip via CSV/Parquet without data loss.
- Test that manifest contains all required fields and that checksums match the actual files.

Logging

- Log file sizes written and confirm the manifest has been written. Log warnings if storing to DuckDB fails.

Configuration schemas

The following YAML schemas define the structure of the configuration files. Example values are provided for illustration.

`config/config.yaml`

```yaml
# Offline-first market monitor configuration
offline: true  # force offline mode (no network requests)
run:
  top_n: 15                      # number of symbols to highlight in report
  conservative: true             # choose conservative scoring variant
paths:
  data_dir: "./data"
  output_dir: "./outputs/runs"
  model_dir: "./models"
  text_corpus_dir: "./data/text_corpus"
macro:
  lookback_years: 5
  zscore_thresholds:
    low: -1.0
    high: 1.0
  series:
```

```yaml
    - name: "term_spread"
      file: "macro/term_spread.csv"
    - name: "cpi_yoy"
      file: "macro/cpi_yoy.csv"
    - name: "vix"
      file: "macro/vix.csv"
themes:
  theme_weights:
    etf: 0.6
    keywords: 0.3
    sector: 0.1
scoring:
  gates:
    min_price_above_sma200: true
    min_adv20_dollar: 5_000_000   # USD
    max_zero_volume_fraction: 0.05
    min_market_cap: 300_000_000   # USD
  weights_conservative:
    return_6m: 0.25
    return_12m: 0.25
    momentum: 0.15
    volatility: -0.15
    drawdown: -0.1
    liquidity: 0.2
  weights_opportunistic:
    return_1m: 0.3
    return_3m: 0.3
    momentum: 0.2
    volatility: -0.1
    drawdown: -0.1
    liquidity: 0.1
  theme_bonus: 0.05  # bonus added per unit of theme confidence
regime_overlay:
  contraction:
    volatility_penalty: 1.2  # multiply volatility weight
    liquidity_bonus: 1.3
  expansion:
    return_bonus: 1.2
    momentum_bonus: 1.1
analogs:
  embedding_model: "sentence-transformers/all-MiniLM-L6-v2"
  chunk_size: 512
  chunk_overlap: 50
  top_k: 3                        # number of analogs per symbol

report:
  include_scenario_section: true
  include_analog_section: true
```

`config/sources.yaml`

This file describes where each dataset lives and how often it should be refreshed. It helps the provisioning script know which resources to fetch when online.

```yaml
symbols:
  nasdaq:
    path: "./data/raw/symbols/nasdaqlisted.txt"
    update_frequency: "monthly"
    source_url: "https://www.nasdaqtrader.com/dynamic/SymDir/nasdaqlisted.txt"
    license: "public domain"
  nyse:
    path: "./data/raw/symbols/nyse.csv"
    update_frequency: "monthly"
    source_url: "https://example.com/nyse_symbols.csv"
    license: "public domain"
ohlcv:
  provider: "stooq"
  base_url: "https://stooq.com/q/d/l/"
  filename_pattern: "{symbol}.csv"
  update_frequency: "daily"
  cache_dir: "./data/raw/ohlcv"
macro:
  term_spread:
    source: "FRED"
    path: "./data/raw/macro/term_spread.csv"
    refresh: "weekly"
    url: "https://fred.stlouisfed.org/series/T10Y2Y"
    license: "public domain"
  vix:
    source: "CBOE"
    path: "./data/raw/macro/vix.csv"
    refresh: "daily"
    url: "https://cdn.cboe.com/api/global/us_indices/daily_prices/VIX.csv"
    license: "CBOE policy"
etf_holdings:
  path: "./data/raw/etf_holdings"
  refresh: "monthly"
edgar:
  path: "./data/raw/edgar"
  refresh: "quarterly"
text_corpus:
  path: "./data/text_corpus/raw"
  refresh: "semiannual"
  notes: "Must contain only public-domain or licensed documents"
```

`config/watchlists.yaml`

Defines theme buckets with seed tickers, relevant ETFs and allowed sectors. Additional themes can be added by editing this file.

```yaml
themes:
  Defense:
    seed_tickers: ["LMT", "GD", "NOC", "RTX"]
    etfs: ["XAR", "ITA"]
    sectors: ["Industrials", "Aerospace & Defense"]
    keywords: ["missile", "defense contract", "military"]
  StrategicTech:
    seed_tickers: ["NVDA", "AMD", "TSM", "ASML"]
    etfs: ["SMH", "AIQ"]
    sectors: ["Information Technology"]
    keywords: ["AI", "semiconductor", "machine learning"]
  CriticalMaterials:
    seed_tickers: ["ALB", "SQM", "LTHM"]
    etfs: ["LIT", "REMX"]
    sectors: ["Materials"]
    keywords: ["lithium", "rare earth", "battery"]
  Energy:
    seed_tickers: ["XOM", "CVX", "COP", "BP"]
    etfs: ["XLE", "VDE"]
    sectors: ["Energy"]
    keywords: ["oil", "gas", "renewable"]
universe_constraints:
  max_universe_size: 3000
  include_etfs: false  # whether ETF tickers themselves are scored
```

`config/logging.yaml`

```yaml
version: 1
formatters:
  simple:
    format: "%(asctime)s [%(levelname)s] %(name)s – %(message)s"
handlers:
  console:
    class: logging.StreamHandler
    level: INFO
    formatter: simple
    stream: ext://sys.stdout
  file:
    class: logging.FileHandler
    level: DEBUG
```

```
    formatter: simple
    filename: "logs/market_app.log"
loggers:
  market_app:
    level: INFO
    handlers: [console, file]
    propagate: false
root:
  level: WARNING
  handlers: [console]
```

Dataset provisioning plan

Required datasets

Dataset	Purpose	Required?	Provisioning steps
Symbols lists	defines the investable universe; includes tickers, names, exchanges	✔	Use `sources.yaml` to download symbol directories (NASDAQ, NYSE, AMEX). Save to `data/raw/symbols/`. Refresh monthly when online.
OHLCV price history	computes returns, volatility, drawdowns	✔	Use free sources like Stooq or Yahoo via `yfinance`. A provisioning script (`provision_data.ps1`) should accept a list of tickers and download CSVs into `data/raw/ohlcv/`. When offline, the pipeline must use the cached CSVs.
Macro series	macro/regime context (term spread, inflation, VIX, etc.)	✔	Download from FRED, CBOE or other providers while online. Cache as CSV in `data/raw/macro/`. Refresh weekly/monthly.
ETF holdings	theme classification via ETFs	Optional	When online, fetch holdings using an API (e.g., ETF.com or IEX) and cache in `data/raw/etf_holdings/`. Update monthly.
EDGAR filings	optional text evidence for themes	Optional	Use the SEC's full-text search or `sec-edgar-downloader` library to download recent 10-K and 10-Q filings. Store raw filings under `data/raw/edgar/` and parse offline. Refresh quarterly.

Dataset	Purpose	Required?	Provisioning steps
Text corpus	historical/ geopolitical analog retrieval	✔ (for analog)	Curate a public-domain corpus (e.g., FOMC minutes, U.S. Fed speeches, Congressional Research Service reports, historical war analyses). The provisioning script should copy or download these texts into `data/text_corpus/raw/`. Acquire only documents with explicit open licences.
Embedding model	transforms corpus and queries into vectors	✔	Download a sentence-transformer model (e.g., `all-MiniLM-L6-v2`) from HuggingFace during provisioning; store under `models/embeddings/`. Only one download is needed; offline runs should not attempt to fetch new models.

Update cadence and offline fallback

Each dataset entry in `sources.yaml` specifies a `refresh` cadence. The provisioning script reads this and decides whether to attempt an update. When the network is unavailable or offline mode is enabled, the script should skip updates and continue using cached files. A `last_refreshed` timestamp can be stored in the metadata to verify staleness.

Data provenance and licensing

For every external file downloaded, create a JSON metadata file in `data/raw/metadata/` with fields:

```
{
  "dataset": "nasdaq_symbols",
  "provider": "NASDAQ Trader",
  "retrieval_date": "2025-12-31",
  "source_url": "https://www.nasdaqtrader.com/dynamic/SymDir/nasdaqlisted.txt",
  "license": "public domain",
  "checksum": "<sha256-of-file>"
}
```

The pipeline should read these metadata files and include them in the run manifest. This ensures compliance with data licences.

Minimum viable corpus

To start, collect roughly 100–200 public-domain documents covering economic crises, wars and geopolitical events. Suitable sources include:

- U.S. Federal Reserve speeches and minutes (public domain).
- Congressional Research Service reports.
- Historical news articles (pre-1923) available via Library of Congress.

- Academic papers on past market crashes (with proper licences).

During provisioning, convert PDFs to text and remove copyrighted material. Document IDs and metadata (title, date, source) must be stored.

Evaluation and backtest plan

Walk-forward backtesting procedure

1. **Universe construction** – For each evaluation date (e.g., the first business day of every month), create the investable universe using data available up to that date. Include symbols that were active at that time to reduce survivorship bias.
2. **Feature computation** – Compute features using only data up to the evaluation date (rolling windows). Avoid using future price data (no look-ahead).
3. **Scoring and ranking** – Apply gating and scoring rules with the configuration fixed at the start of the backtest. Rank symbols by total score.
4. **Evaluation window** – Measure forward returns over a horizon (e.g., next 1 month or 3 months) and record metrics: average return of top N, dispersion, drawdown and turnover (percentage of names replaced relative to previous period).
5. **Regime splits** – Partition backtest periods by macro regime (expansion vs contraction, high vs low inflation) determined using macro series up to the evaluation date. Compare the ranking performance in different regimes.
6. **Leakage controls** – Use point-in-time data. For OHLCV, include delisted stocks by preserving historical symbol files. A survivorship bias article notes that excluding failed or delisted assets can overstate annual returns by 1–4% and underestimate risks [7] ; thus, include all tickers even if they later disappear.
7. **Friction modelling** – Subtract a slippage/spread penalty from forward returns. For example, subtract 0.1% plus half of the bid-ask spread proxy (which can be approximated as `volatility20 / sqrt(252)` times a factor). Also limit the weight of names with low ADV to reduce unrealistic allocations.
8. **Stability diagnostics** – Compute Spearman correlation of scores month-to-month to evaluate rank stability. Higher turnover may indicate noisy scores.
9. **Calibration diagnostics** – Evaluate the monotonicity of scores versus forward returns (e.g., quintile analysis). Inspect whether top-ranked bucket outperforms bottom bucket. Investigate sensitivity of results to changes in weight parameters.

Metrics and outputs

- **Average forward return** of top-`N` symbols vs equal-weight benchmark (e.g., S&P 500 constituents). Do not interpret as profit; this is for monitoring only.
- **Hit rate**: percentage of periods where the top bucket outperforms the benchmark.
- **Maximum drawdown** of the monitored basket. Use the same definition of MDD as in feature engineering: the peak-to-trough decline during the evaluation period [4] .
- **Sharpe ratio** (ex-ante) for the ranking; note that survivorship bias can inflate Sharpe ratios by up to 0.5 points [8] —an important caveat.

Avoiding survivorship and look-ahead biases

- Build a **point-in-time universe**: at each evaluation date, derive the list of active tickers from the historical symbol directories. Do not use present-day lists.
- Include **delisted and bankrupt stocks** in the dataset. Excluding them can make backtest results unrealistically positive; survivorship bias can overstate annual returns by 1–4% [7].
- When computing features, ensure that rolling windows use only data prior to the evaluation date.
- Do **not** tune weights on the whole backtest. Instead, use nested cross-validation: calibrate weights on a training period and test on a hold-out period.

Realistic frictions proxy

Because this application is monitoring-only, it does not execute trades, but backtests should model realistic constraints to assess whether signals would hold after costs. The friction model can include:

- **Slippage**: assume an adverse price move of 0.05–0.1% when entering or exiting a position.
- **Bid-ask spread**: estimate spreads as a function of volatility and liquidity (e.g., `spread ≈ volatility20 × 0.01 / ADV20$`).
- **Illiquidity penalty**: penalize names with `adv20_dollar` below a threshold; they should contribute less to the monitored basket.

Acceptance tests

The following acceptance criteria define when the application can be considered complete.

1. **One-command execution** – Running `.\u200brun.ps1 -Config .\config\config.yaml` from the repository root produces a new folder `outputs/runs/<timestamp>/` containing:

2. `universe.csv` (universe after gating),

3. `classified.csv` (theme classification with confidence and evidence),
4. `features.csv` (raw and normalized features),
5. `eligible.csv` (symbols passing gates),
6. `scored.csv` (scores and flags),
7. `regime.json` (macro regime summary),
8. `report.md` (Markdown report), and

9. `manifest.json` (run manifest with config hash, git SHA, environment, dataset checksums).

10. **Determinism** – Re-running the command with the same inputs and configuration produces identical output files, confirmed by equal file checksums and matching manifest entries.

11. **Offline success** – When the computer's network is disabled or `offline: true`, the pipeline completes using only cached data and pre-downloaded models. Any attempt to access the internet must raise an exception, captured in the logs.

12. **Smoke tests** – Running `pytest` in a clean virtual environment executes unit tests and a small end-to-end pipeline on synthetic data. Invariants such as non-negative `adv20_dollar`, proper z-score calculation, and presence of required columns must pass.

13. **Documentation adequacy** – The files `docs/architecture.md` and `docs/data_provenance.md` must describe module interactions, dependencies, configuration, data provisioning steps and licensing. A new engineer should be able to understand and reproduce the pipeline without additional information.

Implementation roadmap

Phase 1 – Minimum viable product (MVP)

1. **Repository scaffold** – Set up the directory structure, create configuration templates and minimal documentation. Implement logging configuration.
2. **Ingestion** – Develop functions to load symbol lists and OHLCV data from local CSV caches. Implement data provenance tracking. Validate that offline mode prohibits network calls. Optionally implement ETF and EDGAR loaders later in the phase.
3. **Feature engineering** – Compute price/volume and risk/liquidity features as described. Incorporate quality checks and normalization. Store results in CSV/Parquet.
4. **Theme classification (basic)** – Use a static mapping based on ETF memberships and sector classifications from `watchlists.yaml`. Implement confidence scores and store classification results.
5. **Scoring and gating** – Implement hard filters, base scoring using configurable weights, regime overlay with simple rule-based macro states, and risk flags. Support conservative and opportunistic variants.
6. **Reporting** – Create a Markdown report generator that summarises top candidates, macro regime and risk flag distribution. Provide basic explanation packs (features and flags), without analog retrieval yet.
7. **Storage layer** – Implement CSV/Parquet writers and manifest creation. Provide a PowerShell script `run.ps1` that orchestrates the modules and writes outputs. Ensure determinism via seeding and config hashing.
8. **Testing** – Write unit tests for all modules and a smoke test for an end-to-end run using synthetic data.

Deliverables at the end of Phase 1: the app runs offline, computes features, performs gating and scoring, writes a report and manifest, and passes unit/smoke tests.

Phase 2 – Version 2 enhancements

1. **Advanced analog retrieval** – Implement the text corpus ingestion and indexing pipeline. Download and embed the historical/geopolitical corpus, build FAISS and BM25 indexes. Use hybrid search to retrieve top-`k` analog documents for each top-ranked symbol. Elastic's guide notes that hybrid search combines lexical precision with semantic understanding to deliver better results [6]; implement reciprocal rank fusion to merge BM25 and vector results.

2. **Regime modelling upgrades** – Replace simple z-score thresholds with more sophisticated techniques, such as k-means clustering or Hidden Markov Models on macro indicators. Calibrate regime definitions on historical data and store models in `models/regime/`.
3. **Theme confidence improvements** – In addition to ETF and sector mapping, use EDGAR keyword extraction and TF-IDF weighting to refine theme assignments. Provide explanation packs showing top keywords and their frequencies.
4. **Optional ML ranking** – After the scoring system has been validated, consider training a machine-learning model (e.g., gradient boosting) to combine features and macro signals. Use nested cross-validation and ensure no look-ahead. Store the model under `models/scoring/`. Because of the monitoring-only constraint, the model should rank rather than predict returns.
5. **Scenario sensitivity overlays** – Define scenario stress matrices (e.g., war, energy crisis, inflation shock) and compute how scores would change under each scenario. Include this information in the report.
6. **Extended evaluation** – Expand the backtest to multiple decades and more macro regimes. Incorporate survivorship-free constituent lists. Validate that top-ranked baskets have reasonable turnover and drawdowns.
7. **Documentation and examples** – Update `docs/architecture.md` to reflect new modules. Provide example run notebooks illustrating how to interpret analog evidence and macro regimes.

Upon completion of Phase 2, `market_app` will be a robust, explainable, offline-first market monitoring system that produces defensible rankings, risk flags and historical analog evidence. The system will operate entirely on locally cached data after provisioning, fulfilling the offline requirement. The hybrid search, advanced regime modelling and refined theme classification deliver greater insight and explanatory power without crossing into trading or recommendation territory.

---

1 Understanding the 200-Day SMA: Key Indicator for Market Trends
https://www.investopedia.com/ask/answers/013015/why-200-simple-moving-average-sma-so-common-traders-and-analysts.asp

2 Calculate Stock Volatility in Excel: A Step-by-Step Guide
https://www.investopedia.com/ask/answers/021015/how-can-you-calculate-volatility-excel.asp

3 Understanding Downside Risk in Investments: Definition and Calculation
https://www.investopedia.com/terms/d/downsiderisk.asp

4 Maximum drawdown: the formula | Robeco Global
https://www.robeco.com/en-int/insights/2024/10/the-formula-maximum-drawdown

5 Northstar Risk: Average Daily Volume
https://www.northstarrisk.com/average-daily-volume

6 A Comprehensive Hybrid Search Guide | Elastic
https://www.elastic.co/what-is/hybrid-search

7 8 Survivorship Bias in Backtesting Explained
https://www.luxalgo.com/blog/survivorship-bias-in-backtesting-explained/