The App’s coding instructions

Purpose

This page is a **developer-facing build spec** for implementing Operator 1 as a **Kaggle-ready Python project**.

The output of the implementation must be:

- A **2-year daily cache** for a user-selected company.
- A **2-year daily cache** for all linked entities (competitors, supply chain, institutions, sector/industry peers, etc.).
- A unified, point-in-time dataset that is ready for:
    - **Survival mode analysis** (company + country).
    - Downstream temporal modeling modules.
    - Report generation via Gemini (later).

No code is written here. This is a step-by-step instruction document.

---

1) Kaggle execution requirements (must follow)

1.1 Notebook settings

- Turn **Internet = ON** in Kaggle notebook settings.
- Use **Kaggle Secrets** for all API keys.

1.2 Secrets (required)

Store these secrets in Kaggle:

- `EULERPOOL_API_KEY`
- `GEMINI_API_KEY`
- `FMP_API_KEY` (for OHLCV candles)

World Bank Open Data is public and usually does **not** require an API key.

If you later use a gateway, proxy, or premium mirror, store:

- `WORLD_BANK_API_KEY` (optional)

1.3 Reading secrets in Python

- Use Kaggle’s secrets client.
- Never hardcode keys.
- Fail fast with a clear error if any key is missing.

1.4 Packages

- Use `requests` for HTTP.
- Use `pandas`, `numpy` for dataframes and features.
- Use a time/date library (Python stdlib `datetime` is enough).
- Use `pyarrow` (optional) for Parquet caching.

---

2) High-level pipeline (what the code must do)

Step 0 — User-friendly input (start here)

The notebook must start with a simple input cell (no UI framework required) that collects:

- `target_isin` (required)
  - Eulerpool ISIN for the target company. The user finds this outside the notebook.
- `fmp_symbol` (required)
  - FMP trading symbol used to fetch OHLCV (for example `AAPL`). The user finds this outside the notebook.

Rules:

- **Do not ask for company name or country as user inputs.**
- **Country must be taken from Eulerpool profile** after ISIN verification.
- Fail fast if the ISIN cannot be verified via Eulerpool profile.

Step 0.1 — Verify identifiers (strict)

Goal: make identifier mismatches impossible to miss.

1. Verify Eulerpool identity:
   - `GET /api/1/equity/profile/{target_isin}`

    - If this fails, stop immediately (invalid ISIN or API key issue).
2. Verify FMP symbol is usable:
    - Call FMP `quote` for the provided `fmp_symbol`.
    - If this fails, stop immediately (invalid symbol or API key issue).

> ⚠️ We intentionally removed Gemini company-name resolution. The user provides identifiers directly (Eulerpool ISIN + FMP symbol). This removes cross-provider ambiguity and prevents silent cache poisoning.

Implementation:

```
import json
import requests
from difflib import SequenceMatcher

def gemini_generate_search_plan(company_name: str,
                                 company_country: str,
                                 company_hint_ticker: str,
                                 gemini_api_key: str) -> dict:
    """
    Use Gemini to generate intelligent search plan for any company globally.

    Returns:
        dict with search_terms, country_codes, ticker_hints, sector_hints
    """

    prompt = f"""You are a financial data search assistant. A user wants to find a company in the Eulerpool financial database.

User provided:
- Company name: "{company_name}"
- Country hint: "{company_country if company_country else
'unknown'}"
- Ticker hint: "{company_hint_ticker if company_hint_ticker
else 'none'}"

Your task: Generate a comprehensive search plan to find thi
s company in Eulerpool.

IMPORTANT RULES:
1. Do NOT assume the company exists or use prior knowledge
about specific companies
2. Generate search variations that would work for ANY compa
ny name globally
3. Consider international naming conventions (e.g., "Inc",
"Ltd", "GmbH", "SA", "公司", "AG", "NV", "SpA")
4. If country is "unknown", suggest multiple likely countri
es based on company name patterns
5. Output ONLY valid JSON, no additional text

Output JSON schema:

  "primary_search_terms": ["list of 3-5 search strings to t
ry first"],
  "alternative_spellings": ["list of 2-4 alternative name f
ormats"],
  "likely_country_codes": ["list of 1-3 ISO-2 country code
s, or ['GLOBAL'] if truly unknown"],
  "likely_ticker_symbols": ["list of possible ticker format
s, or [] if unknown"],
  "sector_hints": ["list of 1-3 possible sectors based on n
ame, or [] if unclear"],
  "search_strategy": "exact_match_first_then_fuzzy"



Generate the search plan now."""

    # Call Gemini API
    url = f"https://generativelanguage.googleapis.com/v1bet
```

```
a/models/gemini-pro:generateContent?key={gemini_api_key}"

    payload = {
        "contents": [{"parts": [{"text": prompt}]}],
        "generationConfig": {"temperature": 0.2, "maxOutput
Tokens": 1000}
    }

    response = requests.post(url, json=payload)
    response.raise_for_status()

    result = response.json()
    gemini_text = result["candidates"][0]["content"]["part
s"][0]["text"]

    # Extract JSON from Gemini response (may have markdown
code blocks)
    if "```json" in gemini_text:
        gemini_text = gemini_text.split("```json")[1].split
("```")[0]
    elif "```" in gemini_text:
        gemini_text = gemini_text.split("```")[1].split("``
`")[0]

    search_plan = json.loads(gemini_text.strip())

    return search_plan


def search_eulerpool_company(search_term: str, eulerpool_ap
i_key: str) -> list:
    """
    Search Eulerpool for companies matching search_term.

    Returns:
        list of dicts with company info (name, isin, ticke
r, country, sector, etc.)
    """
```

```
    # Eulerpool search endpoint (adjust to actual Eulerpool
API)
    url = f"https://api.eulerpool.com/api/search"

    headers = {"Authorization": f"Bearer {eulerpool_api_ke
y}"}
    params = {"query": search_term, "type": "equity", "limi
t": 10}

    response = requests.get(url, headers=headers, params=pa
rams)
    response.raise_for_status()

    results = response.json()

    # Normalize to standard format
    normalized = []
    for item in results.get('data', []):
        normalized.append({
            'name': item.get('name'),
            'isin': item.get('isin'),
            'ticker': item.get('ticker'),
            'exchange': item.get('exchange'),
            'country': item.get('country'),
            'sector': item.get('sector'),
            'industry': item.get('industry'),
        })

    return normalized


def score_company_match(candidate: dict,
                        search_plan: dict,
                        user_country_hint: str,
                        user_ticker_hint: str) -> float:
    """
    Score how well a candidate matches the search criteria.
    NO HARDCODED WEIGHTS - uses proportional logic.

    Returns:
        float score 0-100
    """

    score = 0.0

    # Ticker match (strongest signal if available)
    if user_ticker_hint and candidate.get('ticker'):
        if candidate['ticker'].upper() == user_ticker_hint.
upper():
            score += 50
        elif user_ticker_hint.upper() in candidate['ticke
r'].upper():
            score += 30

    # Country match
    if user_country_hint and user_country_hint.lower() !=
'unknown':
        if candidate.get('country'):
            # Flexible country matching (US = USA = United
States)
            country_normalized = candidate['country'].upper
()[:2]
            hint_normalized = user_country_hint.upper()[:2]
            if country_normalized == hint_normalized:
                score += 30

    # Name similarity (using Levenshtein-like ratio)
    if candidate.get('name'):
        max_name_similarity = 0
        for search_term in search_plan.get('primary_search_
terms', []):
            similarity = SequenceMatcher(None,
                                         candidate['nam
e'].lower(),
                                         search_term.lower
```

```
()).ratio()
            max_name_similarity = max(max_name_similarity,
similarity)

        score += max_name_similarity * 20

    # Sector hint match
    if candidate.get('sector') and search_plan.get('sector_
hints'):
        for sector_hint in search_plan['sector_hints']:
            if sector_hint.lower() in candidate['sector'].l
ower():
                score += 10
                break

    return score


def resolve_company_with_gemini(company_name: str,
                                 company_country: str,
                                 company_hint_ticker: str,
                                 eulerpool_api_key: str,
                                 gemini_api_key: str) -> di
ct:
    """
    Complete company resolution workflow using Gemini + Eul
erpool.

    Returns:
        dict with resolved company info, or raises exceptio
n if not found
    """

    print(f"\n🔍 Resolving company: {company_name}")

    # Step 1: Generate search plan with Gemini
    print("  📋 Generating search plan with Gemini...")
    search_plan = gemini_generate_search_plan(
        company_name, company_country, company_hint_ticker,
gemini_api_key
    )

    print(f"  Primary search terms: {search_plan['primary_s
earch_terms']}")
    print(f"  Likely countries: {search_plan['likely_countr
y_codes']}")

    # Step 2: Search Eulerpool with all search terms
    all_candidates = []

    search_terms = (search_plan['primary_search_terms'] +
                   search_plan.get('alternative_spellings',
[]))

    for search_term in search_terms:
        try:
            results = search_eulerpool_company(search_term,
eulerpool_api_key)
            all_candidates.extend(results)
        except Exception as e:
            print(f"  ⚠ Search failed for '{search_term}':
{e}")

    if not all_candidates:
        raise ValueError(f"No companies found in Eulerpool
for '{company_name}'")

    # Step 3: Score and rank candidates
    scored_candidates = []
    for candidate in all_candidates:
        score = score_company_match(
            candidate, search_plan, company_country, compan
y_hint_ticker
        )
        scored_candidates.append((score, candidate))
    # Sort by score descending
    scored_candidates.sort(key=lambda x: x[0], reverse=Tru
e)

    # Step 4: Present top matches for user confirmation
    print("\n 📊 Top matches found:")
    top_matches = scored_candidates[:5]

    for i, (score, candidate) in enumerate(top_matches, 1):
        print(f"\n  {i}. {candidate['name']}")
        print(f"     Ticker: {candidate.get('ticker', 'N/
A')}")
        print(f"     ISIN: {candidate.get('isin', 'N/A')}")
        print(f"     Country: {candidate.get('country', 'N/
A')}")
        print(f"     Exchange: {candidate.get('exchange',
'N/A')}")
        print(f"     Sector: {candidate.get('sector', 'N/
A')}")
        print(f"     Match score: {score:.1f}/100")

    # Step 5: User confirms selection
    while True:
        choice = input("\n  Select company (1-5) or 0 to ca
ncel: ")
        try:
            choice_idx = int(choice)
            if choice_idx == 0:
                raise ValueError("User cancelled company se
lection")
            if 1 <= choice_idx <= len(top_matches):
                selected = top_matches[choice_idx - 1][1]
                break
            else:
                print("  Invalid selection, try again.")
        except ValueError as e:
            if "User cancelled" in str(e):
                raise
            print("  Invalid input, please enter a numbe
r.")

    print(f"\n  ✅ Selected: {selected['name']} ({selected
['isin']})")

    return selected
```

Step 0.2 — World Bank indicator mapping (config-driven, optionally Gemini-assisted)

Purpose: Map the resolved country to World Bank indicator codes dynamically (or via config) with no hardcoded country-specific values in code.

Implementation:

```
def gemini_generate_world_bank_indicator_plan(country: str,
                                              country_code_
iso2: str,
                                              gemini_api_ke
y: str) -> dict:
    """
    Use Gemini to propose World Bank indicator codes for an
y country dynamically.
Note: Prefer a checked-in config file for stability. Gemini
output should be treated as a *suggestion* that can update
the config (human-reviewed) rather than being used blindly.

    Returns:
        dict mapping canonical variable names to World Bank
indicator codes
    """

    prompt = f"""You are a macro data mapping assistant. Yo
u need to find World Bank Open Data indicator codes for a s
pecific country.

Resolved country: "{country}" (ISO-2 code: "{country_code_i
so2}")
```

```
Required macro variables (canonical names, World Bank-backe
d):
1. inflation_rate_yoy - Consumer price inflation (annual %)
2. cpi_index - CPI index level
3. unemployment_rate - Unemployment (% of labor force)
4. gdp_growth - GDP growth (annual %)
5. gdp_current_usd - GDP (current US$) for protection logic
6. official_exchange_rate_lcu_per_usd
7. current_account_balance_pct_gdp
8. reserves_months_of_imports
Rates note:
- World Bank does not provide a universal daily policy rate
series.
- Use best-effort proxies (real interest rate, lending rat
e, deposit rate) when available, otherwise keep missing fla
gs.

IMPORTANT RULES:
1. Do NOT use hardcoded series IDs - discover them based on
country
2. For each variable, suggest the most likely World Bank in
dicator code for this country
3. If a series is not available for this country, return nu
ll
4. Prefer globally-available World Bank indicators. Do not
special-case the USA with domestic series IDs.
5. For other countries, use OECD or international series wh
en available
6. Output ONLY valid JSON, no additional text

World Bank notes:
- World Bank indicator codes look like `FP.CPI.TOTL.ZG` (in
flation), `NY.GDP.MKTP.CD` (GDP current US$), `SL.UEM.TOTL.
ZS` (unemployment), etc.
- Frequency is often annual (sometimes monthly). Align to d
aily using as-of logic.
```

```
Output JSON schema:
{
  "country_code_iso2": "{country_code_iso2}",
  "country_name": "{country}",
  "world_bank_indicator_map":
    "inflation_rate_yoy": "INDICATOR_CODE or null",
    "cpi_index": "INDICATOR_CODE or null",
    "unemployment_rate": "INDICATOR_CODE or null",
    "gdp_growth": "INDICATOR_CODE or null",
    "gdp_current_usd": "INDICATOR_CODE or null",
    "official_exchange_rate_lcu_per_usd": "INDICATOR_CODE o
r null",
    "current_account_balance_pct_gdp": "INDICATOR_CODE or n
ull",
    "reserves_months_of_imports": "INDICATOR_CODE or null"
  ,
  "notes": "Brief explanation of any missing series or prox
ies used"
}}

Generate the World Bank indicator mapping now."""

    # Call Gemini API
    url = f"https://generativelanguage.googleapis.com/v1bet
a/models/gemini-pro:generateContent?key={gemini_api_key}"

    payload = {
        "contents": [{"parts": [{"text": prompt}]}],
        "generationConfig": {"temperature": 0.2, "maxOutput
Tokens": 1000}
    }

    response = requests.post(url, json=payload)
    response.raise_for_status()

    result = response.json()
    gemini_text = result["candidates"][0]["content"]["part
s"][0]["text"]
```

```
    # Extract JSON
    if "```json" in gemini_text:
        gemini_text = gemini_text.split("```json")[1].split
("```")[0]
    elif "```" in gemini_text:
        gemini_text = gemini_text.split("```")[1].split("``
`")[0]

    fred_mapping = json.loads(gemini_text.strip())

    return fred_mapping


def fetch_world_bank_indicator(country_iso3: str, indicator
_code: str,
                               start_year: int, end_year: i
nt,
                               timeout_s: int = 30,
                               max_retries: int = 5) -> pd.
Series:
    """
    Fetch a single World Bank indicator time series.
Returns a pandas Series indexed by period end date (usually
yearly; sometimes monthly).

    Returns:
        pandas Series with date index and values
    """

    url = f"https://api.worldbank.org/v2/country/{country_i
so3}/indicator/{indicator_code}"

    params = {
    'format': 'json',
    'per_page': 20000,
    'date': f"{start_year}:{end_year}",
}  # World Bank is typically annual; align later using as-o
```

```
f rules

    response = requests.get(url, params=params)
    response.raise_for_status()

    data = response.json()

    # Parse observations
    dates = []
    values = []

        # World Bank JSON response is usually: [metadata, d
ata_rows]
    rows = data[1] if isinstance(data, list) and len(data)
> 1 else []
    for obs in rows:
        try:
            dates.append(pd.to_datetime(obs['date']))
            values.append(float(obs['value']))
        except (ValueError, KeyError):
            # Skip invalid observations
            continue

    series = pd.Series(values, index=dates, name=indicator_
code)

    return series



def bootstrap_world_bank_macro(country: str,
                               country_code_iso2: str,
                               start_date: str,
                               end_date: str,
                               indicator_map: dict) -> pd.D
ataFrame:
    """
    Complete World Bank macro bootstrap workflow using an i
ndicator map.
```

```
- Convert ISO-2 → ISO-3 using the World Bank country endpoi
nt (cache this).
- Fetch each indicator.
- Align to daily using as-of logic.

    Returns:
        DataFrame with all available macro variables (missi
ng = NaN)
    """

    print(f"\n🌍 Bootstrapping World Bank macro for {countr
y} ({country_code_iso2})")

    # Step 1: Load indicator map from config (preferred)
# Optionally: use Gemini to suggest indicator codes, then s
tore them in config (human-reviewed)
print("  📋 Loading World Bank indicator map...")
    # If using Gemini here, it should propose *World Bank i
ndicator codes*.
    # In production, prefer a checked-in config file that i
s human-reviewed.
    # indicator_map should be passed in (canonical_var -> i
ndicator_code).
    # (No Gemini call is required at runtime.)

    print(f"  Mapped {len([v for v in indicator_map.values
() if v])} indicators")
    # Optional: print notes if your config stores them
    # print(indicator_map.get('notes', ''))

    # Step 2: Fetch each indicator from World Bank
    wb_data = {}

    for var_name, indicator_code in indicator_map.items():
        if not indicator_code:
            print(f"  ⚠ {var_name}: No indicator availabl
e")
            wb_data[var_name] = None
            continue

        try:
            print(f"  📥 Fetching {var_name} ({indicator_co
de})...")
            series = fetch_world_bank_indicator(country_iso
3, indicator_code, start_year, end_year)
            wb_data[var_name] = series
            print(f"    ✅ {len(series)} observations")
        except Exception as e:
            print(f"    ❌ Failed: {e}")
            wb_data[var_name] = None

    # Step 3: Combine into DataFrame (daily index)
    date_range = pd.date_range(start=start_date, end=end_da
te, freq='D')
    wb_df = pd.DataFrame(index=date_range)

    for var_name, series in wb_data.items():
        if series is not None:
            # Forward-fill to daily (World Bank series are
often annual/monthly)
            wb_df[var_name] = series.reindex(date_range, me
thod='ffill')
        else:
            wb_df[var_name] = np.nan

    # Step 4: Add missing flags
    for var_name in wb_data.keys():
        wb_df[f'is_missing_{var_name}'] = wb_df[var_name].i
sna().astype(int)

    print(f"\n  ✅ FRED bootstrap complete")
    print(f"     Available: {sum([1 for v in fred_data.valu
es() if v is not None])}/{len(fred_data)} variables")

    return fred_df
```

Step A — Resolve the user's target company

Input from user: `company_name` (string) plus `company_country` (hint).

1. Call Eulerpool search endpoint(s) to find best matches.
2. Present top matches (name + ticker + ISIN + exchange + country) to user.
3. User confirms the correct company.
4. Store the confirmed identifiers:
    - `target_isin`
    - `target_ticker`
    - `target_country`
    - `target_sector`, `target_industry`

Step B — Build the relationship graph (fully automatic linked entity discovery)

Purpose: Discover and resolve all linked entities (competitors, suppliers, customers, etc.) with NO user input and NO hardcoded data.

Implementation:

```
def gemini_discover_relationships(company_info: dict, gemini_api_key: str) -> dict:
    """
    Use Gemini to discover related entities for any company globally.

    Returns:
        dict with relationship categories and search terms
    """

    prompt = f"""You are a financial relationship mapping assistant. You need to identify companies related to a target company.

Target company:
- Name: "{company_info['name']}"
- Country: "{company_info['country']}"
- Sector: "{company_info.get('sector', 'Unknown')}"
- Industry: "{company_info.get('industry', 'Unknown')}"
- Exchange: "{company_info.get('exchange', 'Unknown')}"

Your task: Identify related companies across these relation
ship types:
1. Competitors - Direct competitors in the same industry
2. Suppliers - Key suppliers providing inputs
3. Customers - Major customers purchasing products/services
4. Financial Institutions - Banks, insurers, investors with
significant exposure
5. Logistics - Shipping, distribution, warehousing partners
6. Regulators - Government bodies, regulatory agencies

IMPORTANT RULES:
1. Do NOT assume specific companies - generate searchable d
escriptions
2. Focus on relationship patterns (e.g., "major smartphone
manufacturers" not "Apple, Samsung")
3. Consider global supply chains - relationships may cross
borders
4. If a relationship type is not applicable, return empty l
ist
5. For each entity, provide search hints to find them in Eu
lerpool
6. Output ONLY valid JSON, no additional text

Output JSON schema:
{{
  "relationships": {{
    "competitors": [

        "search_term": "generic description or company typ
e",
        "country_hint": "ISO-2 code or GLOBAL",
        "sector_hint": "expected sector",
        "reasoning": "why this is a competitor"
    ],
    "suppliers": [...],
    "customers": [...],
    "financial_institutions": [...],
    "logistics": [...],
    "regulators": [...]
  }},
  "relationship_count":
    "competitors": N,
    "suppliers": N,
    "customers": N,
    "financial_institutions": N,
    "logistics": N,
    "regulators": N
  ,
  "discovery_confidence": "high" or "medium" or "low"
}}

Generate the relationship map now. For each category, provi
de 3-7 entities if applicable, empty list if not."""

    # Call Gemini API
    url = f"https://generativelanguage.googleapis.com/v1bet
a/models/gemini-pro:generateContent?key={gemini_api_key}"

    payload = {
        "contents": [{"parts": [{"text": prompt}]}],
        "generationConfig": {"temperature": 0.3, "maxOutput
Tokens": 2000}
    }

    response = requests.post(url, json=payload)
    response.raise_for_status()

    result = response.json()
    gemini_text = result["candidates"][0]["content"]["part
s"][0]["text"]
```

```
    # Extract JSON
    if "```json" in gemini_text:
        gemini_text = gemini_text.split("```json")[1].split
("```")[0]
    elif "```" in gemini_text:
        gemini_text = gemini_text.split("```")[1].split("``
`")[0]

    relationships = json.loads(gemini_text.strip())

    return relationships


def auto_match_entity_to_eulerpool(search_term: str,
                                     country_hint: str,
                                     sector_hint: str,
                                     eulerpool_api_key: st
r) -> dict:
    """
    Automatically match a search term to best Eulerpool res
ult.
    NO user confirmation - uses scoring threshold.

    Returns:
        dict with matched entity or None if no good match
    """

    # Search Eulerpool
    try:
        candidates = search_eulerpool_company(search_term,
eulerpool_api_key)
    except Exception as e:
        print(f"    ⚠ Search failed: {e}")
        return None

    if not candidates:
        return None
```

```
    # Score each candidate
    scored = []
    for candidate in candidates:
        score = 0.0

        # Country match
        if country_hint and country_hint.upper() != 'GLOBA
L':
            if candidate.get('country'):
                if candidate['country'].upper()[:2] == coun
try_hint.upper()[:2]:
                    score += 40

        # Sector match
        if sector_hint and candidate.get('sector'):
            if sector_hint.lower() in candidate['sector'].l
ower():
                score += 30

        # Name similarity to search term
        if candidate.get('name'):
            similarity = SequenceMatcher(None,
                                          candidate['nam
e'].lower(),
                                          search_term.lower
()).ratio()
            score += similarity * 30

        scored.append((score, candidate))

    # Get best match
    scored.sort(key=lambda x: x[0], reverse=True)
    best_score, best_candidate = scored[0]

    # Threshold: only accept if score >= 70
    if best_score >= 70:
        return {
            'name': best_candidate['name'],
            'isin': best_candidate['isin'],
            'ticker': best_candidate.get('ticker'),
            'country': best_candidate['country'],
            'sector': best_candidate.get('sector'),
            'match_confidence': best_score / 100,
            'search_term': search_term
        }
    else:
        return None


def resolve_all_linked_entities(company_info: dict,
                                  eulerpool_api_key: str,
                                  gemini_api_key: str) -> d
ict:
    """
    Complete linked entity resolution workflow.
    Fully automatic - no user input required.

    Returns:
        dict with resolved relationships
    """

    print(f"\n🔗 Discovering linked entities for {company_i
nfo['name']}")

    # Step 1: Gemini discovers relationships
    print("  📋 Discovering relationships with Gemini...")
    relationships = gemini_discover_relationships(company_i
nfo, gemini_api_key)

    print(f"  Discovered {sum(relationships['relationship_c
ount'].values())} potential entities")
    print(f"  Confidence: {relationships['discovery_confide
nce']}")

    # Step 2: Resolve each entity to Eulerpool
    resolved_relationships = {
        'target_company_isin': company_info['isin'],
        'relationships': {},
        'metadata': {
            'total_searched': 0,
            'total_matched': 0,
            'match_rate': 0.0
        }
    }

    for category, entities in relationships['relationship
s'].items():
        print(f"\n  🔍 Resolving {category} ({len(entitie
s)} entities)...")

        resolved_relationships['relationships'][category] =
[]

        for entity in entities:
            search_term = entity['search_term']
            country_hint = entity.get('country_hint', 'GLOB
AL')
            sector_hint = entity.get('sector_hint', '')

            print(f"    Searching: {search_term}")

            resolved_relationships['metadata']['total_searc
hed'] += 1

            # Auto-match to Eulerpool
            matched = auto_match_entity_to_eulerpool(
                search_term, country_hint, sector_hint, eul
erpool_api_key
            )

            if matched:
                print(f"      ✅ Matched: {matched['name']}
(confidence: {matched['match_confidence']:.2f})")
                matched['relationship_type'] = category
                resolved_relationships['relationships'][cat
egory].append(matched)
                resolved_relationships['metadata']['total_m
atched'] += 1
            else:
                print(f"      ❌ No good match found")

    # Step 3: Fallback - if no competitors found, use secto
r peers
    if not resolved_relationships['relationships'].get('com
petitors'):
        print("\n  🔄 No competitors found, using sector pe
ers as fallback...")

        # Search for companies in same sector + industry
        sector_peers = search_eulerpool_by_sector(
            company_info.get('sector'),
            company_info.get('industry'),
            company_info['country'],
            eulerpool_api_key
        )

        # Filter out the target company itself
        sector_peers = [p for p in sector_peers if p.get('i
sin') != company_info['isin']]

        # Take top 5 by market cap (if available)
        sector_peers = sorted(sector_peers,
                             key=lambda x: x.get('market_ca
p', 0),
                             reverse=True)[:5]

        for peer in sector_peers:
            resolved_relationships['relationships'].setdefa
ult('competitors', []).append({
                'name': peer['name'],
                'isin': peer['isin'],
                'ticker': peer.get('ticker'),
                'country': peer['country'],
                'sector': peer.get('sector'),
                'match_confidence': 0.5,  # Lower confidenc
e for sector peers
                'search_term': f"Sector peer: {peer['nam
e']}",
                'relationship_type': 'competitors'
            })

        print(f"    ✅ Added {len(sector_peers)} sector pee
rs as competitors")

    # Calculate match rate
    total_searched = resolved_relationships['metadata']['to
tal_searched']
    total_matched = resolved_relationships['metadata']['tot
al_matched']

    if total_searched > 0:
        resolved_relationships['metadata']['match_rate'] =
total_matched / total_searched

    print(f"\n  ✅ Resolution complete")
    print(f"     Searched: {total_searched}")
    print(f"     Matched: {total_matched}")
    print(f"     Match rate: {resolved_relationships['metad
ata']['match_rate']:.1%}")

    return resolved_relationships


def search_eulerpool_by_sector(sector: str,
                                industry: str,
                                country: str,
                                eulerpool_api_key: str) ->
list:
    """
    Search Eulerpool for companies in same sector/industry.
    Fallback when Gemini relationship discovery fails.

    Returns:
        list of company dicts
    """

    # Eulerpool filter endpoint (adjust to actual API)
    url = f"https://api.eulerpool.com/api/companies"

    headers = {"Authorization": f"Bearer {eulerpool_api_ke
y}"}
    params = {
        'sector': sector,
        'industry': industry,
        'country': country,
        'limit': 20
    }

    response = requests.get(url, headers=headers, params=pa
rams)
    response.raise_for_status()

    results = response.json()

    # Normalize
    normalized = []
    for item in results.get('data', []):
        normalized.append({
            'name': item.get('name'),
            'isin': item.get('isin'),
            'ticker': item.get('ticker'),
            'country': item.get('country'),
            'sector': item.get('sector'),
            'industry': item.get('industry'),
            'market_cap': item.get('market_cap', 0)
        })

    return normalized
```

Step C — Download raw data from Eulerpool (target + linked)

For **target company** and for **each linked entity** (all resolved ISINs), fetch (strict):

- `GET /api/1/equity/profile/{isin}` (identity + classification, includes country)
- `GET /api/1/equity/quotes/{identifier}` (daily market series for last 2 years; do not rely on this for OHLCV completeness)
- `GET /api/1/equity/incomestatement/{isin}` (full history needed for TTM)
- `GET /api/1/equity/balancesheet/{isin}` (full history needed for TTM)
- `GET /api/1/equity/cashflowstatement/{isin}` (full history needed for TTM)

Relationship seed endpoints (target only, strict):

- `GET /api/1/equity/peers/{isin}` (competitor/peer seed set)
- `GET /api/1/equity/supply-chain/{isin}` (supplier/customer seed sets)

Vanity + stability endpoint (target only, strict):

- `GET /api/1/equity/executives/{isin}` (exec compensation + tenure)

Step C.1 — FMP endpoints (OHLCV + quotes)

FMP base: `https://financialmodelingprep.com/stable`

Daily OHLCV (2-year cache default):

- `GET /historical-price-eod/full?symbol={SYMBOL}&apikey={FMP_API_KEY}`

Optional daily variants:

- Dividend adjusted: `GET /historical-price-eod/dividend-adjusted?symbol={SYMBOL}&apikey=...`
- Non-split adjusted: `GET /historical-price-eod/non-split-adjusted?symbol={SYMBOL}&apikey=...`

Real-time quote (verification + UI header):

- `GET /quote?symbol={SYMBOL}&apikey=...`

Batch quotes (watchlists):

- `GET /batch-quote?symbols={CSV_SYMBOLS}&apikey=...`

Intraday candles (UI zoom, NOT part of the 2-year daily cache):

- `GET /historical-chart/1min?symbol={SYMBOL}&apikey=...`

- `GET /historical-chart/5min?symbol={SYMBOL}&apikey=...`
- `GET /historical-chart/15min?symbol={SYMBOL}&apikey=...`
- `GET /historical-chart/30min?symbol={SYMBOL}&apikey=...`
- `GET /historical-chart/1hour?symbol={SYMBOL}&apikey=...`
- `GET /historical-chart/4hour?symbol={SYMBOL}&apikey=...`

API key rule (mandatory):

- If the URL already has a `?` (for example `?symbol=AAPL`) append `&apikey=...`.
- Otherwise append `?apikey=...`.

> ⚠️ We removed Gemini-based FMP symbol resolution. The user provides `fmp_symbol` directly.

FMP base: `https://financialmodelingprep.com/stable`

Daily OHLCV (2-year cache default):

- `GET /historical-price-eod/full?symbol={SYMBOL}&apikey={FMP_API_KEY}`

Optional daily variants:

- Dividend adjusted: `GET /historical-price-eod/dividend-adjusted?symbol={SYMBOL}&apikey=...`
- Non-split adjusted: `GET /historical-price-eod/non-split-adjusted?symbol={SYMBOL}&apikey=...`

Real-time quote (verification + UI header):

- `GET /quote?symbol={SYMBOL}&apikey=...`

Batch quotes (watchlists):

- `GET /batch-quote?symbols={CSV_SYMBOLS}&apikey=...`

Intraday candles (UI zoom, NOT part of the 2-year daily cache):

- `GET /historical-chart/1min?symbol={SYMBOL}&apikey=...`
- `GET /historical-chart/5min?symbol={SYMBOL}&apikey=...`
- `GET /historical-chart/15min?symbol={SYMBOL}&apikey=...`
- `GET /historical-chart/30min?symbol={SYMBOL}&apikey=...`

- `GET /historical-chart/1hour?symbol={SYMBOL}&apikey=...`
- `GET /historical-chart/4hour?symbol={SYMBOL}&apikey=...`

API key rule (mandatory):

- If the URL already has a `?` (for example `?symbol=AAPL`) append `&apikey=...`.
- Otherwise append `?apikey=...`.

Failure policy:

- If FMP symbol cannot be verified, keep OHLCV as missing and stop only if your downstream modules require it. For Op1, OHLCV is required for chart/candlestick modules, so prefer failing fast for the *target company* with a clear message: "Unable to resolve verified FMP symbol for OHLCV".

Step D — Build the 2-year daily cache (point-in-time)

For each entity, create a daily table for the last 2 years:

- Market series is daily.
- Statements are periodic; align them to daily using **as-of** logic.

Step E — Compute derived features (decision + linked variables)

Compute decision variables from the entity's own data.

Compute linked variables by aggregating across relationship sets and peer sets.

Step F — Survival mode analysis readiness

For each day in the 2-year window, the dataset must contain enough features to evaluate:

- Company survival mode
- Country survival mode
- Country protection status
- Vanity expenditure percentage

Store explicit daily flags and metrics:

- `company_survival_mode_flag`
- `country_survival_mode_flag`

- `country_protected_flag`
- `vanity_percentage`
- `survival_hierarchy_weights` (5-element array for Tier 1-5 weights)

Step G — Export artifacts

Persist to disk in Kaggle output:

- `cache/target_company_daily.parquet`
- `cache/linked_entities_daily.parquet`
- `cache/linked_aggregates_daily.parquet`
- `cache/full_feature_table.parquet` (target daily rows enriched with linked variables)
- `cache/metadata.json` (versioning + inputs + identifiers + relationship lists)

---

3) Eulerpool extraction rules (direct vs derived)

Eulerpool does not label "decision" and "linked" variables explicitly. Implement the separation as follows:

IMPORTANT CLARIFICATION: The system will **only pull decision and linked variables** for all entities (target + linked). Unknown variables on any day will be **estimated from known variables** (Sudoku inference). After temporal analysis + burn-out processing, the learned weight patterns will be used to **predict the target company's remaining variables** for next day/week/month/year.

3.1 Direct fields (store as-is)

Profile (direct)

- `isin`, `ticker`, `exchange`, `currency`, `country`, `sector`, `industry`, `sub_industry` (if available)

Quotes (direct, daily)

- `open`, `high`, `low`, `close`, `volume`
- if available: `adjusted_close`, `vwap`, `market_cap`, `shares_outstanding`

Statements (direct, periodic) - DECISION VARIABLES ONLY

Store **only the fields needed to compute decision and linked variables**:

- Revenue, gross profit, EBIT, EBITDA, net income, interest expense, taxes
- Total assets, total liabilities, total equity, current assets, current liabilities
- Cash & equivalents, short-term debt, long-term debt, receivables
- Operating cash flow, capex, investing CF, financing CF, dividends paid

Reason: We focus on decision + linked variables; other variables will be inferred.

3.2 Preprocessing: point-in-time as-of alignment (mandatory)

For each day `t`:

- Choose the latest statement period `p` such that `report_date <= t`.
- Attach statement values to day `t` using canonical `*_asof` names.

This prevents look-ahead bias.

3.3 Derived decision variables (formulas)

Compute these daily for every entity (target + linked):

Returns & risk

- `return_1d`
- `log_return_1d`
- `volatility_21d`
- `drawdown_252d`

Solvency / leverage (Solvency Filter)

- `total_debt_asof = short_term_debt_asof + long_term_debt_asof` (or use a provided total debt field)
- `debt_to_equity = total_debt_asof / total_equity_asof`
- `net_debt = total_debt_asof - cash_and_equivalents_asof`
- `net_debt_to_ebitda = net_debt / ebitda_ttm_asof`

Liquidity / survival signals

- `current_ratio = current_assets_asof / current_liabilities_asof`
- `quick_ratio` (if components exist)
- `cash_ratio = cash_and_equivalents_asof / current_liabilities_asof`

Cash reality (Cash is King Filter)

- `free_cash_flow = operating_cash_flow_asof - capex_asof`
- `free_cash_flow_ttm_asof` (sum last 4 quarters, point-in-time)
- `fcf_yield = free_cash_flow_ttm_asof / market_cap[t]`

Profitability quality

- `gross_margin`, `operating_margin`, `net_margin`, `roe`

Valuation (optional)

- `pe_ratio_calc`, `earnings_yield_calc`, `ps_ratio_calc`, `enterprise_value`, `ev_to_ebitda`

3.4 Derived linked variables (from linked entity sets)

For each day `t`, compute aggregates for each relationship group:

- Competitors aggregates
- Supply chain aggregates (suppliers + customers)
- Financial institutions aggregates

Examples:

- `competitors_avg_return_21d`
- `competitors_median_vol_21d`
- `supply_chain_avg_drawdown_252d`

Also compute peer-group aggregates:

- Sector aggregates
- Industry aggregates

And company-relative versions:

- `rel_strength_vs_sector`
- `valuation_premium_vs_industry`

---

4) Relationship expansion: “find all companies with linked variables”

After Gemini/internal graph produces relationship lists, the code must:

1. **Resolve every related company to Eulerpool**

- If ISIN is missing, use Eulerpool search.
- Apply scoring:
  - exact ticker match > name similarity > same country > same industry.
- If still ambiguous, queue a user confirmation.

2. **Fetch Eulerpool data for each linked company**
   - Same extraction steps as target (profile/quotes/statements).
3. **Preprocess each linked company exactly like the target**
   - Build daily as-of aligned statements.
   - Compute the same derived decision variables.
4. **Aggregate linked variables into the target's daily table**
   - For each day `t`, join competitor/supply-chain/institution aggregates.

---

5) World Bank Open Data integration (country survival + purchasing power)

Eulerpool provides the company's `country`.

Use the **World Bank Open Data API** to fetch country-level macro indicators. Most indicators are **annual** (some are monthly), so they must be aligned to the daily cache using **as-of** logic.

5.0 Canonical macro variables the code must support (World Bank)

A) Purchasing power / inflation (required)

- `inflation_rate_yoy` (annual %)
- `cpi_index` (index level)
- Derived:
  - `inflation_rate_daily_equivalent` (from YoY)
  - `real_return_1d = nominal_return_1d - inflation_rate_daily_equivalent`

B) FX and external stress (recommended)

- `official_exchange_rate_lcu_per_usd`
- `reserves_months_of_imports`
- `current_account_balance_pct_gdp`

C) Real economy health (recommended)

- `unemployment_rate`
- `gdp_growth`
- `gdp_current_usd` (used for protection logic)

D) Interest rate proxies (best-effort)

World Bank does not provide a universal daily policy rate.

- `real_interest_rate` (proxy)
- `lending_interest_rate` (proxy)
- `deposit_interest_rate` (proxy)

If missing, keep `null` + missing flags and continue.

Implementation requirement:

- Use `config/world_bank_indicator_map.yml` to map `canonical_variable -> indicator_code`.
- Implement ISO mapping via the World Bank countries endpoint:
    - resolve Eulerpool ISO-2 → World Bank ISO-3 once, cache it.
- If any canonical variable cannot be fetched for the country, keep it as `null` + missing flag and continue.

Add reliability requirements:

- `requests.get(..., timeout=timeout_s)`
- Retry with exponential backoff on 429/5xx
- Respect `Retry-After` when present
- Cache successful responses on disk to avoid repeated calls in Kaggle

---

5.1 Purchasing Power (inflation-adjusted real return)

Required outputs in target daily table:

- `inflation_rate_yoy`
- `real_return_1d = nominal_return_1d - inflation_rate_daily_equivalent`

5.1 Purchasing Power (inflation-adjusted real return)

Required outputs in target daily table:

- `inflation_rate_yoy`
- `real_return_1d = nominal_return_1d - inflation_rate_daily_equivalent`

Implementation notes:

- Inflation is often monthly YoY; convert carefully to a daily equivalent (document the method).
- Store both the raw inflation value and the conversion method metadata.

5.2 Country protection flag

Your logic requires:

- `country_protected_flag`

Developer requirement:

- Implement this as a configurable rule set that maps country → *macro variables* → thresholds.
- Use World Bank GDP as the default macro input:
  - `gdp_current_usd` (indicator) for the "too big to fail" threshold.
- Keep it data-driven in a YAML/JSON config so it can be revised without rewriting code.

5.3 Country survival mode flag

Compute `country_survival_mode_flag` daily/weekly using macro stress indicators (config-driven).

---

6) Survival mode implementation (company + country + vanity)

6.1 Company Survival Mode Detection

Required decision variables:

- Liquidity: `current_ratio`, `cash_ratio`, `cash_and_equivalents_asof`
- Leverage: `debt_to_equity`, `net_debt_to_ebitda`
- Cash reality: `free_cash_flow_ttm_asof`, `fcf_yield`
- Market stress: `volatility_21d`, `drawdown_252d`

Detection logic (developer must implement):

```
def detect_company_survival_mode(cache: pd.DataFrame) -> p
d.Series:
    """
    Returns boolean series indicating company survival mode
per day

    Triggers when ANY of these critical thresholds are brea
ched:
    - current_ratio < 1.0 (can't pay short-term bills)
    - debt_to_equity > 3.0 (dangerously leveraged)
    - fcf_yield < 0 (burning cash)
    - drawdown_252d < -0.40 (lost >40% of value)
    """

    survival_triggers = (
        (cache['current_ratio'] < 1.0) |
        (cache['debt_to_equity'] > 3.0) |
        (cache['fcf_yield'] < 0) |
        (cache['drawdown_252d'] < -0.40)
    )

    return survival_triggers.astype(int)
```

6.2 Country Survival Mode Detection

Required macro variables (World Bank-backed, best-effort):

- `credit_spread` - Banking stress indicator
- `unemployment_rate` - Economic health

- `yield_curve_slope` - Recession predictor
- `fx_volatility` - Currency crisis indicator
- `policy_rate` - Emergency policy signal

Detection logic:

```
def detect_country_survival_mode(cache: pd.DataFrame, country_code: str) -> pd.Series:
    """
    Returns boolean series indicating country survival mode per day

    Triggers when macro indicators show severe stress:
    - credit_spread > 5.0% (credit market frozen)
    - unemployment_rate increased >3% in 6 months (recession)
    - yield_curve_slope < -0.5% (severe inversion)
    - fx_volatility > 20% annualized (currency crisis)
    """

    # Calculate 6-month unemployment change
    unemp_change_6m = cache['unemployment_rate'] - cache['unemployment_rate'].shift(126)

    survival_triggers = (
        (cache['credit_spread'] > 5.0) |
        (unemp_change_6m > 3.0) |
        (cache['yield_curve_slope'] < -0.5) |
        (cache['fx_volatility'] > 20.0)
    )

    return survival_triggers.astype(int)
```

6.3 Country Protection Flag

Protection logic:

```
def detect_country_protection(cache: pd.DataFrame,
                                sector: str,
                                market_cap: float,
                                country_gdp: float) -> pd.S
eries:
    """
    Returns boolean series indicating if company is protect
ed by government

    Protected if:
    - Strategic sector (defense, energy, banking, utilitie
s)
    - Too big to fail (market_cap > 0.1% of GDP)
    - Emergency policies active (policy_rate cut >2% in 3 m
onths)
    """

    # Strategic sectors that governments typically protect
    STRATEGIC_SECTORS = ['Defense', 'Energy', 'Banking', 'U
tilities', 'Telecommunications']

    is_strategic = sector in STRATEGIC_SECTORS
    is_systemic = market_cap > (country_gdp * 0.001)  # 0.
1% of GDP

    # Check for emergency policy cuts
    policy_rate_cut_3m = cache['policy_rate'].shift(63) - c
ache['policy_rate']
    emergency_policy = policy_rate_cut_3m > 2.0

    protected = (
        is_strategic |
        is_systemic |
        (emergency_policy & is_strategic)  # Emergency + st
rategic = definitely protected
    )
    return protected.astype(int)
```

6.4 Vanity Percentage Calculation

Required additional data (must extract from statements):

- `executive_compensation` (from proxy statements or notes)
- `sga_expenses` (Selling, General & Administrative)
- `stock_buybacks` (from cash flow statement)
- `marketing_expenses` (subset of SG&A if available)

Implementation:

```
def compute_vanity_percentage(cache: pd.DataFrame,
                              linked_caches: Dict,
                              relationships: Dict,
                              industry: str) -> pd.Serie
s:
    """
    Compute vanity expenditure as % of revenue

    Vanity = wasteful spending that signals status, not val
ue creation
    """

    # Component 1: Executive compensation excess
    # Assume exec comp should be max 5% of net income
    exec_comp_threshold = cache['net_income_asof'] * 0.05
    exec_comp_excess = np.maximum(0, cache['executive_compe
nsation'] - exec_comp_threshold)

    # Component 2: SG&A bloat (vs industry median)
    # Compute industry median SG&A ratio from linked entiti
es
    industry_peers = [isin for isin, c in linked_caches.ite
ms()
                      if c.attrs.get('industry') == industr
y]
```

```
    industry_sga_ratios = []
    for isin in industry_peers:
        peer_cache = linked_caches[isin]
        if 'sga_expenses' in peer_cache.columns:
            sga_ratio = peer_cache['sga_expenses'] / peer_c
ache['revenue_asof']
            industry_sga_ratios.append(sga_ratio.median())

    industry_median_sga = np.median(industry_sga_ratios) if
industry_sga_ratios else 0.15

    sga_threshold = cache['revenue_asof'] * industry_median
_sga * 1.2  # 20% above median
    sga_bloat = np.maximum(0, cache['sga_expenses'] - sga_t
hreshold)

    # Component 3: Stock buybacks while burning cash
    vanity_buybacks = np.where(
        cache['free_cash_flow_ttm'] < 0,
        np.maximum(0, cache['stock_buybacks']),
        0
    )

    # Component 4: Excessive marketing during survival mode
    marketing_threshold = cache['revenue_asof'] * 0.10
    marketing_excess = np.where(
        cache['company_survival_mode_flag'] == 1,
        np.maximum(0, cache['marketing_expenses'] - marketi
ng_threshold),
        0
    )

    # Total vanity expenditure
    vanity_expenditures = (
        exec_comp_excess +
        sga_bloat +
        vanity_buybacks +
        marketing_excess
    )

    # Vanity as % of revenue
    vanity_percentage = (vanity_expenditures / cache['reven
ue_asof']) * 100
    vanity_percentage = vanity_percentage.fillna(0).clip(0,
100)

    return vanity_percentage
```

6.5 Survival Hierarchy Weight Selection

Hierarchy configuration (store in config/survival_hierarchy.yml):

```
hierarchies:
  normal:
    name: "Normal Mode"
    tier1_liquidity: 20
    tier2_solvency: 20
    tier3_market_stability: 20
    tier4_profitability: 20
    tier5_growth_valuation: 20

  company_survival:
    name: "Company Survival Mode"
    tier1_liquidity: 50
    tier2_solvency: 30
    tier3_market_stability: 15
    tier4_profitability: 4
    tier5_growth_valuation: 1

  modified_survival:
    name: "Modified Survival (Country Crisis, Company Healt
hy)"
    tier1_liquidity: 40
    tier2_solvency: 35
    tier3_market_stability: 20
    tier4_profitability: 4
```

```yaml
    tier5_growth_valuation: 1

  extreme_survival:
    name: "Extreme Survival (Both in Crisis)"
    tier1_liquidity: 60
    tier2_solvency: 30
    tier3_market_stability: 10
    tier4_profitability: 0
    tier5_growth_valuation: 0

  vanity_adjusted:
    name: "High Vanity Adjustment"
    tier1_liquidity_bonus: 5
    tier4_profitability_penalty: 2
    tier5_growth_valuation_penalty: 3

variable_tier_mapping:
  tier1:
    - cash_and_equivalents_asof
    - cash_ratio
    - free_cash_flow_ttm
    - operating_cash_flow_asof
  tier2:
    - total_debt_asof
    - debt_to_equity
    - net_debt_to_ebitda
    - interest_coverage
    - current_ratio
  tier3:
    - volatility_21d
    - drawdown_252d
    - volume
  tier4:
    - gross_margin
    - operating_margin
    - net_margin
  tier5:
    - revenue_asof
    - pe_ratio
    - ev_to_ebitda
```

Weight selection logic:

```
def select_hierarchy_weights(company_survival: bool,
                              country_survival: bool,
                              country_protected: bool,
                              vanity_pct: float) -> Dict[st
r, float]:
    """
    Select appropriate hierarchy weights based on survival
state

    Returns dict with tier1-5 weights (must sum to 100)
    """

    # Load config
    with open('config/survival_hierarchy.yml', 'r') as f:
        config = yaml.safe_load(f)

    # Case 1: Normal company, normal country
    if not company_survival and not country_survival:
        weights = config['hierarchies']['normal']

    # Case 2: Survival company, normal country
    elif company_survival and not country_survival:
        weights = config['hierarchies']['company_survival']

    # Case 3: Normal company, survival country (not protect
ed)
    elif not company_survival and country_survival and not
country_protected:
        weights = config['hierarchies']['modified_surviva
l']

    # Case 3b: Normal company, survival country (protected)
    elif not company_survival and country_survival and coun
```

```
try_protected:
        weights = config['hierarchies']['normal']  # Protec
ted = use normal

    # Case 4: Both in survival
    elif company_survival and country_survival:
        weights = config['hierarchies']['extreme_survival']

    # Apply vanity adjustment if needed
    if vanity_pct > 10.0 and (company_survival or country_s
urvival):
        adj = config['hierarchies']['vanity_adjusted']
        weights['tier1_liquidity'] += adj['tier1_liquidity_
bonus']
        weights['tier4_profitability'] = max(0, weights['ti
er4_profitability'] - adj['tier4_profitability_penalty'])
        weights['tier5_growth_valuation'] = max(0, weights
['tier5_growth_valuation'] - adj['tier5_growth_valuation_pe
nalty'])

    return {
        'tier1': weights['tier1_liquidity'],
        'tier2': weights['tier2_solvency'],
        'tier3': weights['tier3_market_stability'],
        'tier4': weights['tier4_profitability'],
        'tier5': weights['tier5_growth_valuation'],
    }
```

6.6 Complete Survival Mode Pipeline

Integration into main cache builder:

```
def compute_survival_mode_features(target_cache: pd.DataFra
me,
                                     linked_caches: Dict,
                                     relationships: Dict,
                                     target_sector: str,
                                     target_industry: str,
                                     target_country: str,
                                     target_market_cap: flo
at) -> pd.DataFrame:
    """
    Complete survival mode analysis
    Adds all survival-related flags and weights to cache
    """

    df = target_cache.copy()

    # Step 1: Detect company survival mode
    df['company_survival_mode_flag'] = detect_company_survi
val_mode(df)
    print(f"  Company survival days: {df['company_survival_
mode_flag'].sum()}")

    # Step 2: Detect country survival mode
    df['country_survival_mode_flag'] = detect_country_survi
val_mode(df, target_country)
    print(f"  Country survival days: {df['country_survival_
mode_flag'].sum()}")

    # Step 3: Detect country protection
    # Need country GDP (fetch from World Bank indicators)
    country_gdp = get_country_gdp(target_country)  # Implem
ent this helper
    df['country_protected_flag'] = detect_country_protectio
n(
        df, target_sector, target_market_cap, country_gdp
    )
    print(f"  Protected days: {df['country_protected_fla
g'].sum()}")

    # Step 4: Compute vanity percentage
    df['vanity_percentage'] = compute_vanity_percentage(
        df, linked_caches, relationships, target_industry
    )
    print(f"  Avg vanity %: {df['vanity_percentage'].mean
():.2f}")
```

```
    # Step 5: Select hierarchy weights for each day
    weights_list = []
    for idx in df.index:
        weights = select_hierarchy_weights(
            company_survival=bool(df.loc[idx, 'company_surv
ival_mode_flag']),
            country_survival=bool(df.loc[idx, 'country_surv
ival_mode_flag']),
            country_protected=bool(df.loc[idx, 'country_pro
tected_flag']),
            vanity_pct=df.loc[idx, 'vanity_percentage']
        )
        weights_list.append(weights)

    # Store as structured columns
    df['hierarchy_tier1_weight'] = [w['tier1'] for w in wei
ghts_list]
    df['hierarchy_tier2_weight'] = [w['tier2'] for w in wei
ghts_list]
    df['hierarchy_tier3_weight'] = [w['tier3'] for w in wei
ghts_list]
    df['hierarchy_tier4_weight'] = [w['tier4'] for w in wei
ghts_list]
    df['hierarchy_tier5_weight'] = [w['tier5'] for w in wei
ghts_list]

    # Step 6: Compute hierarchy regime (for analysis)
    def get_regime_name(row):
        if not row['company_survival_mode_flag'] and not ro
w['country_survival_mode_flag']:
            return 'Normal'
        elif row['company_survival_mode_flag'] and not row
['country_survival_mode_flag']:
            return 'Company Survival'
        elif not row['company_survival_mode_flag'] and row
['country_survival_mode_flag']:
            return 'Country Crisis (Protected)' if row['cou
ntry_protected_flag'] else 'Country Crisis'
        else:
            return 'Extreme Survival'

    df['survival_regime'] = df.apply(get_regime_name, axis=
1)

    print("\n  Regime distribution:")
    print(df['survival_regime'].value_counts())

    return df
```

6.7 Using Hierarchy Weights in Temporal Models

When passing data to temporal analysis modules (Phase 2), the hierarchy weights control:

Feature importance in models:

```
# In HMM, Kalman, etc.
feature_weights = {}
for var in decision_variables:
    tier = get_variable_tier(var)  # Look up from config
    weight = cache[f'hierarchy_tier{tier}_weight']
    feature_weights[var] = weight
```

Loss function weighting:

```
# When predicting next day
def weighted_loss(predictions, actuals, hierarchy_weights):
    tier_losses = []
    for tier in [1, 2, 3, 4, 5]:
        tier_vars = get_tier_variables(tier)
        tier_loss = mse(predictions[tier_vars], actuals[tie
r_vars])
        weighted_tier_loss = tier_loss * hierarchy_weights
[f'tier{tier}']
        tier_losses.append(weighted_tier_loss)
    return sum(tier_losses)
```

Prediction confidence:

```
# Reduce confidence in Tier 4/5 predictions during survival
mode
if survival_mode:
    confidence_multiplier = {
        'tier1': 1.0,  # Full confidence in liquidity predi
ctions
        'tier2': 1.0,  # Full confidence in solvency predic
tions
        'tier3': 0.9,
        'tier4': 0.5,  # Low confidence in profitability pr
edictions
        'tier5': 0.3,  # Very low confidence in growth pred
ictions
    }
```

7) Data quality requirements (must not break backtesting)

7.1 No look-ahead tests

Add checks that fail the pipeline if:

- Any statement value used for day `t` has `report_date > t`.

7.1.1 Numerical safety (division-by-zero / invalid math)

Hard requirement: derived feature engineering must never crash on edge cases.

Implement a shared helper for ratio math.

Rules:

- If denominator is `null`, `0`, or near-zero (|denom| < epsilon), return `null`.
- Always set `is_missing_<var> = 1` when returning `null`.
- Also set `invalid_math_<var> = 1` to distinguish "missing from source" vs "invalid computation".

Apply to at least:

- `debt_to_equity`
- `current_ratio`, `cash_ratio`
- `fcf_yield`
- `interest_coverage`

Negative equity policy:

- Allow equity < 0 as a valid distress signal.
- Compute both:
  - `debt_to_equity_signed = debt / equity`
  - `debt_to_equity_abs = debt / abs(equity)`
- Use `debt_to_equity_abs` for threshold triggers.
- Preserve the signed version for interpretation and model learning.

7.2 Missing data strategy

For every computed feature:

- Store `null` when not computable.
- Store a companion missing flag: `is_missing_<feature>`.

7.4 Estimation / imputation (must be cache-based, efficient, and reliable)

Hard requirement: estimation happens **after** all decision + linked variables have been fetched and the full daily cache has been built.

7.4.1 Inputs

- `full_feature_table` (target + linked aggregates + macro), point-in-time aligned.
- No extra API calls for estimation.

7.4.2 Outputs

For each variable `x`:

- `x_observed` (may be null)
- `x_estimated`

- `x_final`
- `x_source` in {observed, estimated}
- `x_confidence` in [0,1]

7.4.3 Efficient estimator

Use a two-pass approach that is linear-time per day:

1. **Deterministic identity fill** (fast, no models).
2. **Regime-weighted rolling imputer** that trains incrementally:
    - Maintain a small set of per-regime models (e.g., BayesianRidge) per tier.
    - Update models online as you traverse days in temporal analysis.
    - Only fit on observed data up to day `t`.

Reliability rules:

- Never overwrite observed truth.
- Penalize learning loss on estimated dimensions less than observed ones.
- Log coverage: percent estimated per variable and per tier.

7.3 Unit normalization

Eulerpool statement values can be in different units depending on source.

Developer requirement:

- Implement unit normalization (or at minimum store `unit` metadata if Eulerpool provides it).

---

8) Project structure (recommended)

The developer should structure the Kaggle notebook/project into modules:

- `config/`
    - `canonical_fields.yml`
    - `survival_rules.yml`
    - `world_bank_indicator_map.yml`
    - `country_protection_rules.yml`

    - `country_survival_rules.yml`
- `src/`
    - `eulerpool_ client.py`
    - `worldbank_ client.py`
    - `gemini_ client.py` (later)
    - `normalize/` adapters for each endpoint
    - `cache_ builder.py`
    - `feature_ engineering.py`
    - `linked_ aggregates.py`
    - `survival_ mode.py`
    - `validators.py`
- `outputs/cache/`

---

9) What the developer must deliver (definition of done)

The implementation is complete when:

1. A user can input a company name.
2. The system resolves it to an Eulerpool identifier.
3. The system discovers linked companies (Gemini/graph) and resolves them to Eulerpool IDs.
4. The system downloads **decision and linked variables only** from Eulerpool for target + linked entities.
5. The system produces a **2-year daily point-in-time cache** for:
    - Target company decision variables.
    - Linked entities decision variables.
    - Linked aggregates (linked variables).
6. **Sudoku inference**: Unknown variables on any day are estimated from known variables.
7. The system computes and stores survival-mode readiness flags.

8. **Temporal analysis + burn-out** modules process the data and learn weight patterns.
9. **Prediction**: Use learned patterns to predict target company's remaining variables for next (day, week, month, year).
10. All artifacts are exported in Kaggle output.

---

10) Temporal Analysis Implementation (Phase 2-3)

This section provides **complete implementation guidance** for the 20+ mathematical modules that power temporal analysis and burn-out.

10.1) Required Python Libraries

Add to notebook:

```
# Time series & statistics
import statsmodels.api as sm
from statsmodels.tsa.statespace.kalman_filter import Kalman
Filter
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.vector_ar.var_model import VAR
from statsmodels.tsa.api import VAR as VARModel
import arch  # For GARCH models

# Machine learning & deep learning
from sklearn.ensemble import RandomForestRegressor, Gradien
tBoostingRegressor
from sklearn.mixture import GaussianMixture
from xgboost import XGBRegressor
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset

# Regime detection & structural breaks
import ruptures as rpt  # For PELT, BinSeg
from hmmlearn import hmm  # Hidden Markov Models
import pymc as pm  # Bayesian change point detection
```

```
# Causality
from statsmodels.tsa.stattools import grangercausalitytests

# Optimization
from scipy.optimize import differential_evolution
from deap import algorithms, base, creator, tools  # Geneti
c algorithms

# Monte Carlo & sensitivity
from SALib.sample import saltelli
from SALib.analyze import sobol

# Pattern recognition
import talib  # Technical analysis patterns
from scipy.signal import find_peaks
from scipy.fft import fft, fftfreq
import pywt  # Wavelets

print("✅ All temporal analysis libraries loaded")
```

10.2) Module 1-4: Regime Detection & Structural Breaks

Implementation:

```
class RegimeDetector:
    """Unified regime detection using multiple methods"""

    def __init__(self, n_regimes=4):
        self.n_regimes = n_regimes
        self.hmm_model = None
        self.gmm_model = None
        self.breakpoints = []

    def fit_hmm(self, returns: np.ndarray, volatility: np.n
darray):
        """Hidden Markov Model for regime detection"""
        # Prepare features: returns + volatility
        X = np.column_stack([returns, volatility])
        X = X[~np.isnan(X).any(axis=1)]  # Remove NaNs
        # Fit Gaussian HMM
        self.hmm_model = hmm.GaussianHMM(
            n_components=self.n_regimes,
            covariance_type="full",
            n_iter=100
        )
        self.hmm_model.fit(X)

        # Predict regimes
        regimes = self.hmm_model.predict(X)
        regime_probs = self.hmm_model.predict_proba(X)

        return regimes, regime_probs

    def fit_gmm(self, returns: np.ndarray):
        """Gaussian Mixture Model for regime clustering"""
        X = returns.reshape(-1, 1)
        X = X[~np.isnan(X)]

        self.gmm_model = GaussianMixture(
            n_components=self.n_regimes,
            covariance_type='full',
            random_state=42
        )
        self.gmm_model.fit(X)

        regimes = self.gmm_model.predict(X)
        return regimes

    def detect_breakpoints_pelt(self, series: np.ndarray, p
enalty=10):
        """PELT algorithm for structural break detection"""
        # Remove NaNs
        series_clean = series[~np.isnan(series)]

        # PELT algorithm
        algo = rpt.Pelt(model="rbf").fit(series_clean)
        breakpoints = algo.predict(pen=penalty)

        self.breakpoints = breakpoints
        return breakpoints

    def detect_breakpoints_bcp(self, series: np.ndarray):
        """Bayesian Change Point detection"""
        # Simplified BCP using PyMC
        with pm.Model() as model:
            # Change point locations
            tau = pm.DiscreteUniform('tau', lower=0, upper=
len(series)-1)

            # Different means before/after change point
            mu_before = pm.Normal('mu_before', mu=0, sigma=
10)
            mu_after = pm.Normal('mu_after', mu=0, sigma=1
0)

            # Assign mean based on change point
            mu = pm.math.switch(tau >= np.arange(len(serie
s)), mu_before, mu_after)

            # Likelihood
            obs = pm.Normal('obs', mu=mu, sigma=1, observed
=series)

            # Sample
            trace = pm.sample(1000, return_inferencedata=Fa
lse, progressbar=False)

        # Most likely change point
        tau_samples = trace['tau']
        changepoint = int(np.median(tau_samples))

        return [changepoint]

# Usage in main pipeline
```

```
def detect_regimes_and_breaks(cache: pd.DataFrame):
    """Run all regime detection methods"""
    print("\n🔍 Detecting regimes and structural break
s...")

    detector = RegimeDetector(n_regimes=4)

    # Extract features
    returns = cache['return_1d'].values
    volatility = cache['volatility_21d'].values
    close = cache['close'].values

    # HMM regimes
    regimes_hmm, regime_probs = detector.fit_hmm(returns, v
olatility)
    cache['regime_hmm'] = np.nan
    cache.loc[~cache['return_1d'].isna(), 'regime_hmm'] = r
egimes_hmm

    # GMM regimes
    regimes_gmm = detector.fit_gmm(returns)
    cache['regime_gmm'] = np.nan
    cache.loc[~cache['return_1d'].isna(), 'regime_gmm'] = r
egimes_gmm

    # Structural breaks (PELT)
    breakpoints = detector.detect_breakpoints_pelt(close, p
enalty=10)
    print(f"  Detected {len(breakpoints)} structural breaks
at days: {breakpoints}")

    # Mark break days
    cache['structural_break'] = 0
    cache.iloc[breakpoints, cache.columns.get_loc('structur
al_break')] = 1

    # Assign regime labels
    regime_names = {0: 'bull', 1: 'bear', 2: 'high_vol', 3:
```

```
'low_vol'}
    cache['regime_label'] = cache['regime_hmm'].map(regime_
names)

    return cache, detector
```

10.3) Module 5-10: Time Series Forecasting Core

10.X) Model training failure handling (must not crash)

Goal: the pipeline must always produce a usable output even if some models fail.

A) General reliability pattern

- Wrap every model fit in `try/except`.
- On failure, record:
  - `model_failed_<name> = 1`
  - `model_failure_reason_<name>` (string)
- Fall back to a simpler model (below).
- Never stop the notebook due to a single model failure.

B) VAR singularity / unstable estimates

Common causes: multicollinearity, too many variables vs rows, non-stationary scale.

Mitigations (in priority order):

1. **Prune features** (already supported by Granger pruning). Add a hard cap `max_vars_for_var`.
2. **Reduce lag order** until the system is full-rank.
3. **Standardize** variables before VAR.
4. Use **regularized VAR** proxy (practical fallback): fit independent AR(1) per variable if VAR fails.

Fallback implementation contract:

- If VAR fails, produce `var_forecast` via AR(1) per variable (or simple last-value baseline if AR fails).

C) LSTM non-convergence / exploding loss

Mitigations:

- Standardize inputs (already in spec).
- Clip gradients (`torch.nn.utils.clip_grad_norm_`).
- Early stop on plateau.
- Lower learning rate and reduce model size.
- Reduce sequence length when data is thin.

Fallback implementation contract:

- If LSTM fails to converge within `max_epochs` or loss becomes NaN/Inf:
  - fall back to tree-based model (RandomForest / GradientBoosting) or linear model (Ridge) for next-step prediction.
  - keep the ensemble running without LSTM weights.

D) Ensemble robustness

- Ensemble must accept a subset of available models.
- Ensemble weights must re-normalize over the models that succeeded.
- Confidence should decrease when fewer models are available.

E) Minimum viable forecasting fallback (always available)

Even if everything fails, ensure the pipeline can still predict next step:

- `baseline_pred(x[t+1]) = x[t]` (last observation)
- or `EMA` forecast per variable.

This baseline keeps the predict → compare → update loop alive and prevents crashes.

Kalman Filter Implementation:

```
class AdaptiveKalmanFilter:
    """Adaptive Kalman Filter for Tier 1-2 variables"""

    def __init__(self, n_states=5):
        self.n_states = n_states
        self.state = np.zeros(n_states)
        self.P = np.eye(n_states)  # State covariance
        self.Q = np.eye(n_states) * 0.01  # Process noise
        self.R = np.eye(n_states) * 0.1   # Measurement noi
se

    def predict(self, F: np.ndarray, u: np.ndarray = None):
        """Predict next state"""
        # State prediction
        if u is not None:
            self.state = F @ self.state + u
        else:
            self.state = F @ self.state

        # Covariance prediction
        self.P = F @ self.P @ F.T + self.Q

        return self.state

    def update(self, z: np.ndarray, H: np.ndarray):
        """Update with observation"""
        # Innovation
        y = z - H @ self.state

        # Innovation covariance
        S = H @ self.P @ H.T + self.R

        # Kalman gain
        K = self.P @ H.T @ np.linalg.inv(S)

        # State update
        self.state = self.state + K @ y
        # Covariance update
        self.P = (np.eye(self.n_states) - K @ H) @ self.P

        return self.state

def apply_kalman_filter(cache: pd.DataFrame, tier1_vars: li
st):
    """Apply Kalman filter to Tier 1 variables"""
    print("\n🎯 Applying Kalman filter to Tier 1 variable
s...")

    n_vars = len(tier1_vars)
    kf = AdaptiveKalmanFilter(n_states=n_vars)

    # State transition matrix (identity + small drift)
    F = np.eye(n_vars) * 0.99

    # Observation matrix (direct observation)
    H = np.eye(n_vars)

    predictions = []

    for idx in range(len(cache)):
        # Predict
        pred = kf.predict(F)
        predictions.append(pred.copy())

        # Update with observation if available
        obs = cache.loc[cache.index[idx], tier1_vars].value
s
        if not np.isnan(obs).any():
            kf.update(obs, H)

    # Store predictions
    for i, var in enumerate(tier1_vars):
        cache[f'{var}_kalman_pred'] = [p[i] for p in predic
tions]
```

```
    return cache, kf
```

GARCH for Volatility:

```
def fit_garch_model(returns: pd.Series):
    """GARCH(1,1) model for volatility forecasting"""
    from arch import arch_model

    # Remove NaNs
    returns_clean = returns.dropna() * 100  # Scale to perc
entages

    # Fit GARCH(1,1)
    model = arch_model(returns_clean, vol='Garch', p=1, q=
1)
    res = model.fit(disp='off')

    # Forecast volatility
    forecast = res.forecast(horizon=1)
    vol_forecast = np.sqrt(forecast.variance.values[-1, 0])
/ 100

    return res, vol_forecast
```

VAR for Multi-Variable:

```
def fit_var_model(cache: pd.DataFrame, variables: list, lag
s=5):
    """Vector Autoregression for all tiers"""
    print(f"\n📊 Fitting VAR({lags}) model...")

    # Select variables and remove NaNs
    data = cache[variables].dropna()

    # Fit VAR
    model = VAR(data)
    results = model.fit(maxlags=lags, ic='aic')
    print(f"  Optimal lags: {results.k_ar}")

    return results

def predict_var(results, steps=1):
    """Predict multiple steps ahead with VAR"""
    forecast = results.forecast(results.endog[-results.k_a
r:], steps=steps)
    return forecast
```

LSTM Implementation:

```
class LSTMPredictor(nn.Module):
    """LSTM for non-linear pattern capture"""

    def __init__(self, input_size, hidden_size=64, num_laye
rs=2, output_size=1):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.lstm = nn.LSTM(input_size, hidden_size, num_la
yers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # x shape: (batch, seq_len, input_size)
        lstm_out, _ = self.lstm(x)
        # Take last timestep
        out = self.fc(lstm_out[:, -1, :])
        return out

def train_lstm(cache: pd.DataFrame, variables: list, seq_le
ngth=20, epochs=50):
    """Train LSTM on historical data"""
    print(f"\n🧠 Training LSTM (seq_length={seq_length}, ep
ochs={epochs})...")
```

```
    # Prepare data
    data = cache[variables].dropna().values

    # Normalize
    from sklearn.preprocessing import StandardScaler
    scaler = StandardScaler()
    data_scaled = scaler.fit_transform(data)

    # Create sequences
    X, y = [], []
    for i in range(len(data_scaled) - seq_length):
        X.append(data_scaled[i:i+seq_length])
        y.append(data_scaled[i+seq_length])  # Predict all
variables

    X = np.array(X)
    y = np.array(y)

    # Convert to tensors
    X_tensor = torch.FloatTensor(X)
    y_tensor = torch.FloatTensor(y)

    dataset = TensorDataset(X_tensor, y_tensor)
    loader = DataLoader(dataset, batch_size=32, shuffle=Tru
e)

    # Initialize model
    model = LSTMPredictor(input_size=len(variables), output
_size=len(variables))
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.0
01)

    # Train
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for batch_X, batch_y in loader:
            optimizer.zero_grad()
            outputs = model(batch_X)
            loss = criterion(outputs, batch_y)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        if (epoch + 1) % 10 == 0:
            print(f"  Epoch {epoch+1}/{epochs}, Loss: {tota
l_loss/len(loader):.6f}")

    return model, scaler
```

10.4) Module 11-13: Causality Analysis

```
def compute_granger_causality(cache: pd.DataFrame, variable
s: list, max_lag=5):
    """Granger causality test for variable relationships"""
    print("\n🔗 Computing Granger causality matrix...")

    # Prepare data
    data = cache[variables].dropna()

    # Causality matrix
    causality_matrix = pd.DataFrame(
        np.zeros((len(variables), len(variables))),
        index=variables,
        columns=variables
    )

    # Test each pair
    for i, var1 in enumerate(variables):
        for j, var2 in enumerate(variables):
            if i == j:
                continue

            try:
                # Granger causality test
                test_result = grangercausalitytests(
                    data[[var2, var1]],  # [y, x] -> does x
cause y?
                    maxlag=max_lag,
                    verbose=False
                )

                # Get minimum p-value across lags
                p_values = [test_result[lag+1][0]['ssr_ftes
t'][1] for lag in range(max_lag)]
                min_p = min(p_values)

                # Store significance (1 if p < 0.05)
                causality_matrix.loc[var2, var1] = 1 if min
_p < 0.05 else 0

            except:
                causality_matrix.loc[var2, var1] = 0

    print(f"  Significant causal relationships: {causality_
matrix.sum().sum()}")

    return causality_matrix

def prune_weak_relationships(cache: pd.DataFrame, causality
_matrix: pd.DataFrame, threshold=0.5):
    """Remove variables with weak causal links"""
    # Count incoming causal links per variable
    causal_strength = causality_matrix.sum(axis=1)

    # Keep variables with at least threshold connections
    strong_vars = causal_strength[causal_strength >= thresh
old].index.tolist()

    print(f"  Pruned to {len(strong_vars)} strongly connect
ed variables")
```

```
    return strong_vars
```

10.5) Module 14-16: Ensemble & Optimization

```
class EnsemblePredictor:
    """Ensemble of all prediction modules"""

    def __init__(self, hierarchy_weights: dict):
        self.hierarchy_weights = hierarchy_weights
        self.module_weights = {
            'kalman': 0.3,
            'var': 0.2,
            'lstm': 0.25,
            'rf': 0.15,
            'garch': 0.1
        }
        self.models = {}

    def add_model(self, name: str, model):
        """Register a trained model"""
        self.models[name] = model

    def predict(self, current_state: dict, tier: int) -> tu
ple:
        """Generate ensemble prediction for a variable tie
r"""
        predictions = []
        weights = []

        # Collect predictions from each model
        for model_name, weight in self.module_weights.items
():
            if model_name in self.models:
                pred = self.models[model_name].predict(curr
ent_state)
                predictions.append(pred)
                # Adjust weight by tier
                tier_weight = self.hierarchy_weights[f'tier
{tier}']
                adjusted_weight = weight * (tier_weight / 2
0)  # Normalize to standard
                weights.append(adjusted_weight)

        # Weighted average
        weights = np.array(weights) / np.sum(weights)
        ensemble_pred = np.average(predictions, weights=wei
ghts)

        return ensemble_pred, predictions, weights

def optimize_ensemble_weights_genetic(cache: pd.DataFrame,
                                       variables: list,
                                       models: dict,
                                       generations=50):
    """Use genetic algorithm to optimize ensemble weight
s"""
    print(f"\n🧬 Optimizing ensemble weights (generations=
{generations})...")

    # Define fitness function
    def fitness(weights):
        # Ensure weights sum to 1
        weights = np.array(weights) / np.sum(weights)

        # Compute weighted predictions
        errors = []
        for idx in range(len(cache) - 1):
            # Get predictions from each model
            preds = [model.predict(cache.iloc[idx]) for mod
el in models.values()]

            # Weighted ensemble
            ensemble = np.average(preds, weights=weights)
            # Actual
            actual = cache.iloc[idx + 1][variables].values

            # Error
            error = np.mean((ensemble - actual) ** 2)
            errors.append(error)

        # Return negative MSE (minimize)
        return -np.mean(errors),

    # Genetic algorithm setup
    creator.create("FitnessMin", base.Fitness, weights=(1.
0,))
    creator.create("Individual", list, fitness=creator.Fitn
essMin)

    toolbox = base.Toolbox()
    toolbox.register("attr_float", np.random.random)
    toolbox.register("individual", tools.initRepeat, creato
r.Individual,
                     toolbox.attr_float, n=len(models))
    toolbox.register("population", tools.initRepeat, list,
toolbox.individual)
    toolbox.register("evaluate", fitness)
    toolbox.register("mate", tools.cxBlend, alpha=0.5)
    toolbox.register("mutate", tools.mutGaussian, mu=0, sig
ma=0.2, indpb=0.2)
    toolbox.register("select", tools.selTournament, tournsi
ze=3)

    # Run evolution
    pop = toolbox.population(n=100)
    algorithms.eaSimple(pop, toolbox, cxpb=0.7, mutpb=0.2,
ngen=generations, verbose=False)

    # Best individual
    best = tools.selBest(pop, k=1)[0]
    best_weights = np.array(best) / np.sum(best)
    print(f"  Optimized weights: {dict(zip(models.keys(), b
est_weights))}")

    return best_weights
```

10.6) Module 17-18: Monte Carlo Simulation

```
def regime_aware_monte_carlo(cache: pd.DataFrame,
                               variables: list,
                               regime: str,
                               n_simulations=10000,
                               horizon=1):
    """Monte Carlo simulation conditional on regime"""
    print(f"\n🎲 Running Monte Carlo ({n_simulations} simul
ations, regime={regime})...")

    # Filter data by regime
    regime_data = cache[cache['regime_label'] == regime][va
riables].dropna()

    if len(regime_data) < 10:
        print("  ⚠ Insufficient regime data, using full sam
ple")
        regime_data = cache[variables].dropna()

    # Compute regime-specific statistics
    means = regime_data.mean()
    cov = regime_data.cov()

    # Generate scenarios
    scenarios = np.random.multivariate_normal(means, cov, s
ize=n_simulations)

    # Compute percentiles
    percentiles = np.percentile(scenarios, [5, 50, 95], axi
s=0)
```

```
    return scenarios, percentiles

def importance_sampling_monte_carlo(cache: pd.DataFrame,
                                     survival_threshold: fl
oat,
                                     n_simulations=10000):
    """Importance sampling for tail risk (survival mode eve
nts)"""
    print(f"\n⚠ Running importance sampling Monte Carlo for
tail risk...")

    # Focus on tail events (survival mode)
    # Over-sample low liquidity scenarios

    current_cash_ratio = cache['cash_ratio'].iloc[-1]

    # Proposal distribution: shift mean toward survival thr
eshold
    proposal_mean = survival_threshold * 0.8
    proposal_std = current_cash_ratio * 0.5

    # Sample from proposal
    samples = np.random.normal(proposal_mean, proposal_std,
n_simulations)

    # Compute importance weights
    # (ratio of target density to proposal density)
    target_density = scipy.stats.norm.pdf(samples, loc=curr
ent_cash_ratio, scale=proposal_std)
    proposal_density = scipy.stats.norm.pdf(samples, loc=pr
oposal_mean, scale=proposal_std)
    weights = target_density / proposal_density

    # Weighted survival probability
    survival_prob = np.sum(weights * (samples > survival_th
reshold)) / np.sum(weights)

    print(f"  Estimated survival probability: {survival_pro
b:.4f}")

    return survival_prob, samples, weights
```

10.7) Complete Temporal Analysis Pipeline

```
def run_temporal_analysis_pipeline(cache: pd.DataFrame,
                                     target_info: dict,
                                     hierarchy_weights: dic
t):
    """
    Complete temporal analysis: forward pass + burn-out + p
redictions

    This is the main entry point for Phase 2-3
    """

    print("="*60)
    print("🚀 PHASE 2-3: TEMPORAL ANALYSIS + BURN-OUT")
    print("="*60)

    # Step 1: Detect regimes and structural breaks
    cache, regime_detector = detect_regimes_and_breaks(cach
e)

    # Step 2: Define variable tiers
    tier1_vars = ['cash_ratio', 'free_cash_flow_ttm', 'oper
ating_cash_flow_asof', 'cash_and_equivalents_asof']
    tier2_vars = ['debt_to_equity', 'net_debt_to_ebitda',
'interest_coverage', 'current_ratio']
    tier3_vars = ['volatility_21d', 'drawdown_252d', 'volum
e']
    tier4_vars = ['gross_margin', 'operating_margin', 'net_
margin']
    tier5_vars = ['revenue_asof', 'pe_ratio', 'ev_to_ebitd
a']

    all_vars = tier1_vars + tier2_vars + tier3_vars + tier4
```

```
_vars + tier5_vars

    # Step 3: Granger causality pruning
    causality_matrix = compute_granger_causality(cache, all
_vars, max_lag=5)
    strong_vars = prune_weak_relationships(cache, causality
_matrix, threshold=2)

    # Step 4: Train individual models
    print("\n📚 Training prediction models...")

    # Kalman filter
    cache, kalman_filter = apply_kalman_filter(cache, tier1
_vars)

    # VAR model
    var_model = fit_var_model(cache, strong_vars, lags=5)

    # GARCH for volatility
    garch_model, vol_forecast = fit_garch_model(cache['retu
rn_1d'])

    # LSTM
    lstm_model, lstm_scaler = train_lstm(cache, strong_var
s, seq_length=20, epochs=50)

    # Random Forest
    from sklearn.ensemble import RandomForestRegressor
    rf_model = RandomForestRegressor(n_estimators=100, max_
depth=10, random_state=42)

    # Prepare RF data
    X_rf = cache[strong_vars].iloc[:-1].dropna()
    y_rf = cache[strong_vars].iloc[1:].loc[X_rf.index]
    rf_model.fit(X_rf, y_rf)

    # Step 5: Build ensemble
    ensemble = EnsemblePredictor(hierarchy_weights)
    ensemble.add_model('kalman', kalman_filter)
    ensemble.add_model('var', var_model)
    ensemble.add_model('garch', garch_model)
    ensemble.add_model('lstm', lstm_model)
    ensemble.add_model('rf', rf_model)

    # Step 6: Forward pass training
    print("\n⏩ Running forward pass (day-by-day trainin
g)...")

    errors_by_tier = {1: [], 2: [], 3: [], 4: [], 5: []}

    for t in range(len(cache) - 1):
        current_state = cache.iloc[t]
        actual_next = cache.iloc[t + 1]

        # Predict each tier
        for tier_idx, tier_vars in enumerate([tier1_vars, t
ier2_vars, tier3_vars, tier4_vars, tier5_vars], 1):
            pred, _, _ = ensemble.predict(current_state, ti
er=tier_idx)

            # Compute error
            actual_vals = actual_next[tier_vars].values
            error = np.mean((pred - actual_vals) ** 2)
            errors_by_tier[tier_idx].append(error)

        # Progress
        if t % 100 == 0:
            print(f"  Day {t}/{len(cache)-1}")

    # Step 7: Burn-out process
    print("\n🔥 Running burn-out process (6-month intensive
retraining)...")

    burn_out_results = []
    best_accuracy = float('inf')
    best_patterns = None
    # Last 6 months (~130 trading days)
    burn_out_cache = cache.iloc[-130:].copy()

    for iteration in range(10):
        print(f"\n  Burn-out iteration {iteration + 1}/10")

        # Retrain with higher learning rate
        # (simplified: just retrain LSTM)
        lstm_burnout, _ = train_lstm(burn_out_cache, strong
_vars, seq_length=20, epochs=20)

        # Test on last 20 days
        test_cache = burn_out_cache.iloc[-20:]
        test_errors = []

        for t in range(len(test_cache) - 1):
            # ... prediction logic ...
            pass

        avg_error = np.mean(test_errors) if test_errors els
e float('inf')
        burn_out_results.append(avg_error)

        if avg_error < best_accuracy:
            best_accuracy = avg_error
            best_patterns = lstm_burnout
            print(f"    ✅ New best accuracy: {avg_error:.6
f}")
        else:
            print(f"    Accuracy: {avg_error:.6f} (no impro
vement)")

        # Early stopping if no improvement for 3 iterations
        if len(burn_out_results) >= 3 and all(burn_out_resu
lts[-3:] >= best_accuracy):
            print("    Early stopping")
            break
    # Step 8: Final predictions (next day/week/month/year)
    print("\n🔮 Generating predictions...")

    current_state = cache.iloc[-1]
    current_regime = current_state['regime_label']

    # Next day prediction
    pred_next_day = {}
    for tier_idx, tier_vars in enumerate([tier1_vars, tier2
_vars, tier3_vars, tier4_vars, tier5_vars], 1):
        pred, _, _ = ensemble.predict(current_state, tier=t
ier_idx)
        pred_next_day[f'tier{tier_idx}'] = pred

    # Monte Carlo uncertainty
    scenarios, percentiles = regime_aware_monte_carlo(
        cache, all_vars, current_regime, n_simulations=1000
0
    )

    # Survival probability
    survival_prob, _, _ = importance_sampling_monte_carlo(
        cache, survival_threshold=1.0, n_simulations=10000
    )

    # Step 9: Build complete company profile
    company_profile = build_company_profile(
        cache=cache,
        target_info=target_info,
        predictions=pred_next_day,
        scenarios=scenarios,
        percentiles=percentiles,
        survival_prob=survival_prob,
        hierarchy_weights=hierarchy_weights,
        model_metrics={
            'burn_out_iterations': len(burn_out_results),
            'final_accuracy_tier1': 1 - np.mean(errors_by_t
ier[1][-20:]),
            'final_accuracy_tier2': 1 - np.mean(errors_by_t
ier[2][-20:]),
            'final_accuracy_tier3': 1 - np.mean(errors_by_t
ier[3][-20:]),
            'final_accuracy_tier4': 1 - np.mean(errors_by_t
ier[4][-20:]),
            'final_accuracy_tier5': 1 - np.mean(errors_by_t
ier[5][-20:]),
        }
    )

    print("\n" + "="*60)
    print("✅ TEMPORAL ANALYSIS COMPLETE")
    print("="*60)

    return company_profile

def build_company_profile(cache, target_info, predictions,
scenarios, percentiles,
                           survival_prob, hierarchy_weight
s, model_metrics):
    """Build comprehensive company profile for Gemini"""

    profile = {
        "company": target_info,
        "current_state": {
            "date": cache.index[-1].strftime("%Y-%m-%d"),
            "regime": cache['regime_label'].iloc[-1],
            "survival_flags": {
                "company_survival_mode": bool(cache['compan
y_survival_mode_flag'].iloc[-1]),
                "country_survival_mode": bool(cache['countr
y_survival_mode_flag'].iloc[-1]),
                "country_protected": bool(cache['country_pr
otected_flag'].iloc[-1]),
            },
            "vanity_percentage": cache['vanity_percentag
```

```
e'].iloc[-1],
            "hierarchy_weights": hierarchy_weights,
            # ... all tier variables ...
        },
        "historical": {
            "date_range": [cache.index[0].strftime("%Y-%m-%
d"), cache.index[-1].strftime("%Y-%m-%d")],
            "return_total": (cache['close'].iloc[-1] / cach
e['close'].iloc[0]) - 1,
            # ... more metrics ...
        },
        "predictions_next_day": predictions,
        "model_metrics": model_metrics,
        "survival_probability": survival_prob,
    }

    return profile
```

10.8) Integration Points in Main Workflow

In the main notebook, call temporal analysis after cache building:

```
# After Step 8 from survival mode (cache is complete)
print("\n" + "="*60)
print("🎯 Starting Temporal Analysis Pipeline...")
print("="*60)

# Run complete temporal analysis
company_profile = run_temporal_analysis_pipeline(
    cache=target_cache,
    target_info=target,
    hierarchy_weights={
        'tier1': 20,  # Will be adjusted based on survival
mode
        'tier2': 20,
        'tier3': 20,
        'tier4': 20,
        'tier5': 20,
    }
)

# Save profile
import json
with open('cache/company_profile.json', 'w') as f:
    json.dump(company_profile, f, indent=2, default=str)

print("\n✅ Company profile saved to cache/company_profile.
json")
```

11) Complete Company Profile Builder (Post Burn-Out)

After temporal analysis and burn-out complete, build the comprehensive profile that will be sent to Gemini.

11.1) Profile Building Function

```
def build_complete_company_profile(cache: pd.DataFrame,
                                     linked_caches: Dict[st
r, pd.DataFrame],
                                     macro_cache: pd.DataFr
ame,
                                     target_info: dict,
                                     temporal_results: dic
t,
                                     burn_out_results: dic
t,
                                     relationships: dict) -
> dict:
    """
    Build comprehensive company profile with all variables
needed for Gemini report.

    Args:
        cache: Target company 2-year daily cache
        linked_caches: Dict of {isin: cache} for all linked
entities
```

```
        macro_cache: Country macro variables from World Ban
k Open Data (as-of aligned)
        target_info: Company identity metadata
        temporal_results: Results from temporal analysis pi
peline
        burn_out_results: Results from burn-out process
        relationships: Discovered relationships (competitor
s, supply chain, etc.)

    Returns:
        Complete company profile dict ready for Gemini
    """

    print("\n" + "="*60)
    print("📊 BUILDING COMPLETE COMPANY PROFILE")
    print("="*60)

    # Extract current state (last row)
    current = cache.iloc[-1]
    current_date = cache.index[-1]

    # Extract historical period
    start_date = cache.index[0]
    end_date = cache.index[-1]

    # -----------------------------------------------------
--------------
    # CATEGORY 1: Company Identity
    # -----------------------------------------------------
--------------
    print("\n1️⃣ Extracting company identity...")

    company = {
        "name": target_info.get('name'),
        "ticker": target_info.get('ticker'),
        "isin": target_info.get('isin'),
        "exchange": target_info.get('exchange'),
        "country": target_info.get('country'),
        "sector": target_info.get('sector'),
        "industry": target_info.get('industry'),
        "sub_industry": target_info.get('sub_industry'),
        "currency": target_info.get('currency'),
        "market_cap_current": current.get('market_cap'),
        "shares_outstanding_current": current.get('shares_o
utstanding'),
    }

    # -----------------------------------------------------
--------------
    # CATEGORY 2: Current State Snapshot
    # -----------------------------------------------------
--------------
    print("2️⃣ Building current state snapshot...")

    current_state = {
        "date": current_date.strftime("%Y-%m-%d"),
        "regime": current.get('regime_label'),

        "tier1_liquidity": {
            "cash_and_equivalents": current.get('cash_and_e
quivalents_asof'),
            "cash_ratio": current.get('cash_ratio'),
            "free_cash_flow_ttm": current.get('free_cash_fl
ow_ttm_asof'),
            "operating_cash_flow": current.get('operating_c
ash_flow_asof'),
            "current_ratio": current.get('current_ratio'),
        },

        "tier2_solvency": {
            "total_debt": current.get('total_debt_asof'),
            "debt_to_equity": current.get('debt_to_equit
y'),
            "net_debt": current.get('net_debt'),
            "net_debt_to_ebitda": current.get('net_debt_to_
ebitda'),
            "interest_coverage": current.get('interest_cove
rage'),
            "total_equity": current.get('total_equity_aso
f'),
        },

        "tier3_market_stability": {
            "volatility_21d": current.get('volatility_21
d'),
            "drawdown_252d": current.get('drawdown_252d'),
            "volume_avg_21d": cache['volume'].tail(21).mean
(),
            "beta_252d": current.get('beta_252d'),
            "close_current": current.get('close'),
        },

        "tier4_profitability": {
            "gross_margin": current.get('gross_margin'),
            "operating_margin": current.get('operating_marg
in'),
            "net_margin": current.get('net_margin'),
            "roe": current.get('roe'),
            "roa": current.get('roa'),
            "revenue_ttm": current.get('revenue_ttm_asof'),
            "net_income_ttm": current.get('net_income_ttm_a
sof'),
            "ebitda_ttm": current.get('ebitda_ttm_asof'),
        },

        "tier5_growth_valuation": {
            "revenue_growth_yoy": current.get('revenue_grow
th_yoy'),
            "earnings_growth_yoy": current.get('earnings_gr
owth_yoy'),
            "pe_ratio": current.get('pe_ratio_calc'),
            "earnings_yield": current.get('earnings_yield_c
alc'),
            "ps_ratio": current.get('ps_ratio_calc'),
            "pb_ratio": current.get('pb_ratio'),
            "ev_to_ebitda": current.get('ev_to_ebitda'),
            "enterprise_value": current.get('enterprise_val
ue'),
        },

        "survival": {
            "company_survival_mode": bool(current.get('comp
any_survival_mode_flag')),
            "country_survival_mode": bool(current.get('coun
try_survival_mode_flag')),
            "country_protected": bool(current.get('country_
protected_flag')),
            "survival_regime": current.get('survival_regim
e'),
            "hierarchy_tier1_weight": current.get('hierarch
y_tier1_weight'),
            "hierarchy_tier2_weight": current.get('hierarch
y_tier2_weight'),
            "hierarchy_tier3_weight": current.get('hierarch
y_tier3_weight'),
            "hierarchy_tier4_weight": current.get('hierarch
y_tier4_weight'),
            "hierarchy_tier5_weight": current.get('hierarch
y_tier5_weight'),
        },

        "vanity": {
            "vanity_percentage": current.get('vanity_percen
tage'),
            "exec_comp_excess": current.get('exec_comp_exce
ss'),
            "sga_bloat": current.get('sga_bloat'),
            "vanity_buybacks": current.get('vanity_buyback
s'),
            "marketing_excess": current.get('marketing_exce
ss'),
        },
    }

    # -----------------------------------------------------------
--------------
    # CATEGORY 3: Historical Performance
    # -----------------------------------------------------------
--------------
    print("3️⃣ Computing historical performance metrics...")

    # Total return
    return_total = (cache['close'].iloc[-1] / cache['clos
e'].iloc[0]) - 1
    return_annualized = ((1 + return_total) ** (1/2)) - 1
# 2 years

    # Real return (inflation adjusted)
    if 'real_return_1d' in cache.columns:
        real_return_total = cache['real_return_1d'].sum()
    else:
        real_return_total = return_total  # Fallback if inf
lation not available

    # Risk metrics
    volatility_annualized = cache['return_1d'].std() * np.s
qrt(252)
    max_drawdown = cache['drawdown_252d'].min()

    # Sharpe ratio (assume risk-free rate = 0.03 for simpli
city)
    risk_free_rate = 0.03
    sharpe_ratio = (return_annualized - risk_free_rate) / v
olatility_annualized if volatility_annualized > 0 else 0

    # Return distribution
    up_days = (cache['return_1d'] > 0).sum()
    down_days = (cache['return_1d'] < 0).sum()
    total_days = len(cache)
    historical = {
        "date_range_start": start_date.strftime("%Y-%m-%
d"),
        "date_range_end": end_date.strftime("%Y-%m-%d"),
        "return_total": return_total,
        "return_annualized": return_annualized,
        "return_real": real_return_total,
        "volatility_annualized": volatility_annualized,
        "sharpe_ratio": sharpe_ratio,
        "max_drawdown": max_drawdown,
        "up_days_percentage": (up_days / total_days) * 100,
        "down_days_percentage": (down_days / total_days) *
100,
        "best_day_return": cache['return_1d'].max(),
        "worst_day_return": cache['return_1d'].min(),
    }

    # -------------------------------------------------------
--------------
    # CATEGORY 4: Survival Mode Analysis
    # -------------------------------------------------------
--------------
    print("4️⃣ Analyzing survival mode history...")

    survival_days_company = cache['company_survival_mode_fl
ag'].sum()
    survival_days_country = cache['country_survival_mode_fl
ag'].sum()
    survival_days_both = ((cache['company_survival_mode_fla
g'] == 1) &
                          (cache['country_survival_mode_fla
g'] == 1)).sum()

    # Detect survival episodes
    def count_episodes(series):
        """Count number of distinct episodes (consecutive T
rue values)"""
        episodes = 0
        in_episode = False
        for val in series:
            if val and not in_episode:
                episodes += 1
                in_episode = True
            elif not val:
                in_episode = False
        return episodes

    survival_episodes = count_episodes(cache['company_survi
val_mode_flag'])

    # Vanity interpretation
    vanity_pct = current.get('vanity_percentage', 0)
    if vanity_pct <= 2:
        vanity_interp = "Disciplined - Lean operations"
    elif vanity_pct <= 5:
        vanity_interp = "Normal - Acceptable level"
    elif vanity_pct <= 10:
        vanity_interp = "Elevated - Watch for cuts"
    elif vanity_pct <= 20:
        vanity_interp = "High - Company is bloated"
    else:
        vanity_interp = "Extreme - Survival probability is
low"

    survival_analysis = {
        "survival_days_company": int(survival_days_compan
y),
        "survival_days_country": int(survival_days_countr
y),
        "survival_days_both": int(survival_days_both),
        "survival_episodes_count": survival_episodes,
        "vanity_percentage_current": vanity_pct,
        "vanity_percentage_avg_2y": cache['vanity_percentag
e'].mean(),
        "vanity_interpretation": vanity_interp,
    }
    # ------------------------------------------------------------
--------------
    # CATEGORY 5: Regime Analysis
    # ------------------------------------------------------------
--------------
    print("5️⃣ Analyzing regime distribution...")

    regime_breakdown = cache['regime_label'].value_counts
().to_dict()
    regime_transitions = (cache['regime_label'] != cache['r
egime_label'].shift()).sum() - 1

    structural_breaks = []
    if 'structural_break' in cache.columns:
        break_indices = cache[cache['structural_break'] ==
1].index
        structural_breaks = [idx.strftime("%Y-%m-%d") for i
dx in break_indices]

    regime_analysis = {
        "regime_breakdown": regime_breakdown,
        "regime_current": current.get('regime_label'),
        "regime_transitions_count": int(regime_transition
s),
        "structural_breaks_detected": structural_breaks,
        "structural_breaks_count": len(structural_breaks),
    }

    # ------------------------------------------------------------
--------------
    # CATEGORY 6: Linked Variables
    # ------------------------------------------------------------
--------------
    print("6️⃣ Computing linked variables...")

    # Sector aggregates
    sector = target_info.get('sector')
    industry = target_info.get('industry')

    sector_peers = [c for isin, c in linked_caches.items()
                    if c.attrs.get('sector') == sector]
    industry_peers = [c for isin, c in linked_caches.items
()
                      if c.attrs.get('industry') == industr
y]

    # Compute sector medians (if peers exist)
    if sector_peers:
        sector_median_return = np.median([c['return_21d'].i
loc[-1] for c in sector_peers if 'return_21d' in c.column
s])
        sector_median_vol = np.median([c['volatility_21d'].
iloc[-1] for c in sector_peers if 'volatility_21d' in c.col
umns])
    else:
        sector_median_return = 0
        sector_median_vol = 0

    # Relative strength
    rel_strength = current.get('return_21d', 0) - sector_me
dian_return

    # Industry aggregates
    if industry_peers:
        industry_median_debt = np.median([c['debt_to_equit
y'].iloc[-1] for c in industry_peers if 'debt_to_equity' in
c.columns])
        industry_median_fcf = np.median([c['fcf_yield'].ilo
c[-1] for c in industry_peers if 'fcf_yield' in c.columns])
        industry_median_pe = np.median([c['pe_ratio_calc'].
iloc[-1] for c in industry_peers if 'pe_ratio_calc' in c.co
lumns])
    else:
        industry_median_debt = 0
        industry_median_fcf = 0
        industry_median_pe = 0

    valuation_premium = current.get('pe_ratio_calc', 0) - i
ndustry_median_pe

    # Competitor aggregates
    competitor_isins = relationships.get('competitors', [])
    competitors = [linked_caches[c['isin']] for c in compet
itor_isins if c['isin'] in linked_caches]

    if competitors:
        competitors_avg_return = np.mean([c['return_21d'].i
loc[-1] for c in competitors if 'return_21d' in c.columns])
        competitors_avg_vol = np.mean([c['volatility_21d'].
iloc[-1] for c in competitors if 'volatility_21d' in c.colu
mns])
        competitors_avg_debt = np.mean([c['debt_to_equit
y'].iloc[-1] for c in competitors if 'debt_to_equity' in c.
columns])
    else:
        competitors_avg_return = 0
        competitors_avg_vol = 0
        competitors_avg_debt = 0

    # Macro from World Bank Open Data (as-of aligned daily 
macro cache)
    macro_current = macro_cache.iloc[-1] if len(macro_cach
e) > 0 else {}

    linked_variables = {
        "sector": {
            "median_return_21d": sector_median_return,
            "median_volatility_21d": sector_median_vol,
            "rel_strength_vs_sector": rel_strength,
        },
        "industry": {
            "median_return_21d": 0,  # Compute if needed
            "median_debt_to_equity": industry_median_debt,
            "median_fcf_yield": industry_median_fcf,
            "valuation_premium_vs_industry": valuation_prem
ium,
        },
        "competitors": {
            "count": len(competitors),
            "avg_return_21d": competitors_avg_return,
            "avg_volatility_21d": competitors_avg_vol,
            "avg_debt_to_equity": competitors_avg_debt,
        },
        "macro": {
            "inflation_rate_yoy": macro_current.get('inflat
ion_rate_yoy'),
            "cpi_index": macro_current.get('cpi_index'),
            "official_exchange_rate_lcu_per_usd": macro_cur
rent.get('official_exchange_rate_lcu_per_usd'),
            "unemployment_rate": macro_current.get('unemplo
yment_rate'),
            "gdp_growth": macro_current.get('gdp_growth'),
            "gdp_current_usd": macro_current.get('gdp_curre
nt_usd'),
        },
    }

    # -----------------------------------------------------------
--------------
    # CATEGORY 7: Model Performance
    # -----------------------------------------------------------
--------------
    print("7️⃣ Extracting model performance metrics...")

    model_metrics = {
        "burn_out_iterations": burn_out_results.get('iterat
ions', 0),
        "burn_out_converged": burn_out_results.get('converg
ed', False),
        "final_accuracy_tier1": temporal_results.get('accur
acy_tier1', 0),
        "final_accuracy_tier2": temporal_results.get('accur
acy_tier2', 0),
        "final_accuracy_tier3": temporal_results.get('accur
acy_tier3', 0),
        "final_accuracy_tier4": temporal_results.get('accur
acy_tier4', 0),
        "final_accuracy_tier5": temporal_results.get('accur
acy_tier5', 0),
        "regime_classification_accuracy": temporal_results.
get('regime_accuracy', 0),
        "best_performing_module": temporal_results.get('bes
t_module', 'Ensemble'),
        "ensemble_weights_final": temporal_results.get('ens
emble_weights', {}),
    }

    # -----------------------------------------------------
--------------
    # CATEGORY 8: Predictions
    # -----------------------------------------------------
--------------
    print("8️⃣ Extracting predictions...")

    predictions = temporal_results.get('predictions', {})

    # Apply Technical Alpha protection (mask next-day OHLC
except Low)
    if 'next_day' in predictions and 'ohlc' in predictions
['next_day']:
        predictions['next_day']['ohlc'] = {
            'low': predictions['next_day']['ohlc'].get('lo
w'),
            'high': 'MASKED - Technical Alpha Protection',
            'open': 'MASKED - Technical Alpha Protection',
            'close': 'MASKED - Technical Alpha Protection',
        }

    # -----------------------------------------------------
```

```
--------------
    # CATEGORY 9: Ethical Filters
    # ------------------------------------------------------------
--------------
    print("9️⃣ Computing ethical filter results...")

    # Purchasing Power Filter
    if return_total > 0 and real_return_total > 0:
        pp_verdict = "PASS"
    elif return_total > 0 and real_return_total < 0:
        pp_verdict = "FAIL - Nominal gain, real loss"
    else:
        pp_verdict = "FAIL - Negative returns"

    # Solvency Filter
    debt_eq = current.get('debt_to_equity', 0)
    if debt_eq < 1.0:
        solv_verdict = "PASS - Conservative"
    elif debt_eq < 2.0:
        solv_verdict = "PASS - Stable"
    elif debt_eq < 3.0:
        solv_verdict = "WARNING - Elevated"
    else:
        solv_verdict = "FAIL - Fragile"

    # Gharar Filter
    vol = current.get('volatility_21d', 0)
    if vol < 0.15:
        stability_score = 9
        gharar_verdict = "LOW - Stable"
    elif vol < 0.25:
        stability_score = 7
        gharar_verdict = "MODERATE - Calculated Risk"
    elif vol < 0.40:
        stability_score = 4
        gharar_verdict = "HIGH - Speculation"
    else:
        stability_score = 2
```

```
            gharar_verdict = "EXTREME - Gambling"

        # Cash is King Filter
        fcf_yield = current.get('fcf_yield', 0)
        if fcf_yield > 0.05:
            cash_verdict = "PASS - Strong"
        elif fcf_yield > 0.02:
            cash_verdict = "PASS - Healthy"
        elif fcf_yield > 0:
            cash_verdict = "WARNING - Weak"
        else:
            cash_verdict = "FAIL - Burning Cash"

        filters = {
            "purchasing_power": {
                "verdict": pp_verdict,
                "nominal_return": return_total,
                "real_return": real_return_total,
                "inflation_impact": return_total - real_return_
    total,
            },
            "solvency": {
                "verdict": solv_verdict,
                "debt_to_equity": debt_eq,
                "threshold": 3.0,
            },
            "gharar": {
                "verdict": gharar_verdict,
                "volatility": vol,
                "stability_score": stability_score,
            },
            "cash_is_king": {
                "verdict": cash_verdict,
                "fcf_yield": fcf_yield,
            },
        }

        # -----------------------------------------------------
```

```
--------------
    # Assemble complete profile
    # -------------------------------------------------------
--------------
    print("\n✅ Complete company profile built successfull
y")

    profile = {
        "company": company,
        "current_state": current_state,
        "historical": historical,
        "survival_analysis": survival_analysis,
        "regime_analysis": regime_analysis,
        "linked_variables": linked_variables,
        "predictions": predictions,
        "model_metrics": model_metrics,
        "filters": filters,
    }

    return profile
```

11.2) Usage in Main Pipeline

```
# After temporal analysis and burn-out complete
company_profile = build_complete_company_profile(
    cache=target_cache,
    linked_caches=linked_caches,
    macro_cache=macro_cache,
    target_info=target_company_info,
    temporal_results=temporal_analysis_results,
    burn_out_results=burn_out_results,
    relationships=relationships
)

# Save to JSON
import json
with open('cache/company_profile.json', 'w') as f:
    json.dump(company_profile, f, indent=2, default=str)
```

```
print("\n✅ Company profile saved to cache/company_profile.
json")
```

12) Gemini Report Generation (Phase 4)

12.1) Gemini Report Generator

```
def generate_bloomberg_report_with_gemini(company_profile:
dict,
                                          gemini_api_key:
str) -> str:
    """
    Generate comprehensive Bloomberg-style investment repor
t using Gemini API.

    Args:
        company_profile: Complete company profile dict
        gemini_api_key: Gemini API key from Kaggle secrets

    Returns:
        Markdown-formatted report text
    """

    print("\n" + "="*60)
    print("📄 GENERATING BLOOMBERG-STYLE REPORT WITH GEMIN
I")
    print("="*60)

    # -----------------------------------------------------
--------------
    # Part 1: Build structured prompt
    # -----------------------------------------------------
--------------

    prompt = f"""
You are a Bloomberg-style financial analyst specializing in
```

```
comprehensive equity research.
You have been provided with a complete company profile that
includes:
- 2 years of historical financial and market data
- Advanced temporal analysis using 20+ mathematical models
- Survival mode analysis (company + country)
- Ethical filter assessments (Islamic finance compatible, b
ut universally valuable)
- Multi-horizon predictions with uncertainty quantification
- Regime detection and structural break analysis
- Linked variables (sector, industry, competitors, macro)

Your task is to generate a professional investment report t
hat synthesizes this information
into actionable insights for sophisticated investors.

---

REPORT STRUCTURE (MUST INCLUDE ALL SECTIONS):

1. EXECUTIVE SUMMARY
   - 3 bullet points summarizing key findings
   - Clear investment recommendation: BUY / HOLD / SELL wit
h confidence level (High/Medium/Low)
   - 12-month target price with rationale

2. COMPANY OVERVIEW
   - Company identity and classification
   - Current market position and capitalization
   - Sector and industry context

3. HISTORICAL PERFORMANCE ANALYSIS (2 Years)
   - Total return vs real return (Purchasing Power filter a
pplied)
   - Risk-adjusted performance: Sharpe ratio, maximum drawd
own
   - Regime breakdown: time spent in bull/bear/high-volatil
ity regimes
```

```
   - Structural breaks and major market events detected
   - Up days vs down days distribution

4. CURRENT FINANCIAL HEALTH (Tier-by-Tier Breakdown)

   **Explain the Tier Hierarchy:**
   - Why the 5-tier system matters
   - How weights change in different survival regimes
   - Current hierarchy weights and what they mean

   **Tier 1: Liquidity & Cash (Weight: {company_profile['cu
rrent_state']['survival']['hierarchy_tier1_weight']}%)**
   - Cash and equivalents, cash ratio, free cash flow
   - Cash is King filter results
   - Interpretation: Can the company survive short-term str
ess?

   **Tier 2: Solvency & Debt (Weight: {company_profile['cur
rent_state']['survival']['hierarchy_tier2_weight']}%)**
   - Debt-to-equity, net debt to EBITDA, interest coverage
   - Solvency filter results
   - Interpretation: Is the company overleveraged or stabl
e?

   **Tier 3: Market Stability (Weight: {company_profile['cu
rrent_state']['survival']['hierarchy_tier3_weight']}%)**
   - Volatility, drawdown, volume
   - Gharar filter results (speculation vs calculated risk)
   - Interpretation: How wild is the ride?

   **Tier 4: Profitability (Weight: {company_profile['curre
nt_state']['survival']['hierarchy_tier4_weight']}%)**
   - Margins (gross, operating, net), ROE, ROA
   - Interpretation: Quality of earnings

   **Tier 5: Growth & Valuation (Weight: {company_profile
['current_state']['survival']['hierarchy_tier5_weight']}%)*
*
```

```
   - Revenue/earnings growth, P/E, EV/EBITDA
   - Interpretation: Is the price justified?

5. SURVIVAL MODE ANALYSIS
   - Current survival status:
     - Company survival mode: {company_profile['current_sta
te']['survival']['company_survival_mode']}
     - Country survival mode: {company_profile['current_sta
te']['survival']['country_survival_mode']}
     - Country protection: {company_profile['current_stat
e']['survival']['country_protected']}
   - Current regime: {company_profile['current_state']['sur
vival']['survival_regime']}
   - Historical survival episodes (count and duration)
   - Vanity expenditure analysis:
     - Current vanity percentage: {company_profile['surviva
l_analysis']['vanity_percentage_current']:.2f}%
     - Interpretation: {company_profile['survival_analysi
s']['vanity_interpretation']}
   - What vanity spending reveals about management discipli
ne

6. LINKED VARIABLES & MARKET CONTEXT
   - Sector performance: relative strength vs sector median
   - Industry positioning: valuation premium/discount vs in
dustry
   - Competitor health assessment: how does the company com
pare?
   - Supply chain risk analysis (if applicable)
   - Macro environment:
     - Inflation rate: {company_profile['linked_variables']
['macro'].get('inflation_rate', 'N/A')}
     - Policy rate: {company_profile['linked_variables']['m
acro'].get('policy_rate', 'N/A')}
     - Yield curve slope: {company_profile['linked_variable
s']['macro'].get('yield_curve_slope', 'N/A')}
     - Credit spread: {company_profile['linked_variables']
['macro'].get('credit_spread', 'N/A')}
```

```
   - How macro conditions affect company outlook

7. TEMPORAL ANALYSIS & MODEL INSIGHTS
   - Current market regime: {company_profile['regime_analys
is']['regime_current']}
   - Regime distribution over 2 years
   - Regime transitions and what they signal
   - Structural breaks detected: {company_profile['regime_a
nalysis']['structural_breaks_count']}
   - Model performance summary:
     - Tier 1 accuracy: {company_profile['model_metrics']
['final_accuracy_tier1']:.1%}
     - Tier 2 accuracy: {company_profile['model_metrics']
['final_accuracy_tier2']:.1%}
     - Tier 3 accuracy: {company_profile['model_metrics']
['final_accuracy_tier3']:.1%}
     - Tier 4 accuracy: {company_profile['model_metrics']
['final_accuracy_tier4']:.1%}
     - Tier 5 accuracy: {company_profile['model_metrics']
['final_accuracy_tier5']:.1%}
   - Best performing module: {company_profile['model_metric
s']['best_performing_module']}
   - Confidence levels in predictions

8. PREDICTIONS & FORECASTS

   **Next Day:**
   - NOTE: Due to Technical Alpha protection, only Low pric
e is shown for next-day OHLC
   - Predicted Low: {company_profile['predictions'].get('ne
xt_day', {}).get('ohlc', {}).get('low', 'N/A')}
   - Tier 1-5 variable predictions with uncertainty bands
(5th, 50th, 95th percentiles)
   - Survival probability for next day

   **Next Week:**
   - Expected return and volatility
   - Full OHLC candlestick series
   - Predicted technical patterns

   **Next Month:**
   - Price target range (5th, 50th, 95th percentiles)
   - Key events to watch
   - Regime shift predictions

   **Next Year:**
   - Annual outlook with uncertainty
   - Predicted regime changes by quarter
   - Long-term trajectory

   **Monte Carlo Uncertainty:**
   - Tail risk scenarios (worst 5%)
   - Base case (50th percentile)
   - Upside scenarios (top 5%)

9. TECHNICAL PATTERNS & CHART ANALYSIS
   - Describe the 2-year price chart with regime shading
   - Historical candlestick patterns detected (last 6 month
s)
   - Predicted patterns for next week/month
   - Chart interpretation and technical signals
   - Support and resistance levels

10. ETHICAL FILTER ASSESSMENT

    **Purchasing Power Filter:**
    - Verdict: {company_profile['filters']['purchasing_powe
r']['verdict']}
    - Nominal return: {company_profile['filters']['purchasi
ng_power']['nominal_return']:.2%}
    - Real return: {company_profile['filters']['purchasing_
power']['real_return']:.2%}
    - Interpretation: Did investors actually gain purchasin
g power?

    **Solvency Filter:**
    - Verdict: {company_profile['filters']['solvency']['ver
dict']}
    - Debt-to-equity: {company_profile['filters']['solvenc
y']['debt_to_equity']:.2f}
    - Why this matters: Overleveraged companies are fragile
in downturns

    **Gharar Filter (Uncertainty/Speculation):**
    - Verdict: {company_profile['filters']['gharar']['verdi
ct']}
    - Volatility: {company_profile['filters']['gharar']['vo
latility']:.2%}
    - Stability score: {company_profile['filters']['ghara
r']['stability_score']}/10
    - Is this calculated risk or gambling?

    **Cash is King Filter:**
    - Verdict: {company_profile['filters']['cash_is_king']
['verdict']}
    - FCF yield: {company_profile['filters']['cash_is_kin
g']['fcf_yield']:.2%}
    - Why cash flow matters more than accounting profit

    **Overall Ethical Score:**
    - Combine all filters into an overall assessment
    - Is this investment suitable for ethical/Islamic inves
tors?
    - Universal lessons: Why these filters matter for ALL i
nvestors, not just religious ones

11. RISK FACTORS & LIMITATIONS
    - Model assumptions and their limitations
    - Key risks to the forecast:
      - Company-specific risks
      - Industry/sector risks
      - Macro/country risks
    - Scenarios that could invalidate predictions
    - Data quality and coverage limitations
    - Black swan events not captured by models
11.1 LIMITATIONS (SHORT, REQUIRED)
    Provide a short, clearly labeled limitations section (5
to 10 bullets) covering:
    - Data coverage window (what dates are covered by the 2
-year cache).
    - OHLCV source (FMP) and any known caveats.
    - Macro frequency reality (World Bank indicators are of
ten annual or monthly, aligned daily using as-of logic).
    - Missingness summary (top missing variables, percent m
issing, and where missingness is concentrated).
    - Any modules that failed during this run (if applicabl
e) and how the report compensated.
    This section must be easy for a non-technical client to
understand.

12. INVESTMENT RECOMMENDATION

    **Recommendation:** [BUY / HOLD / SELL]

    **Confidence Level:** [High / Medium / Low]

    **12-Month Target Price:** $XXX

    **Rationale:**
    - Summarize the key factors driving the recommendation
    - Which tiers are most important in this decision?
    - How do survival mode, ethical filters, and prediction
s combine?

    **Key Catalysts to Watch:**
    - Events or metrics that would change the recommendatio
n

    **Entry Strategy:**
    - Recommended entry price or conditions

    **Exit Strategy:**
    - Price targets for taking profits
    - Stop-loss levels

    **Position Sizing:**
    - Suggested portfolio allocation based on risk profile

13. APPENDIX
    - Methodology summary:
      - Temporal analysis modules used (HMM, Kalman, VAR, L
STM, etc.)
      - Burn-out process explanation
      - Ensemble weighting approach
    - Variable tier definitions (Tier 1-5 explained in deta
il)
    - Glossary of technical terms
    - Data sources:
      - Eulerpool (company financials and market data)
      - Country/macros (World Bank Open Data)
      - Gemini (relationship discovery)
    - Data timestamps and coverage
    - Disclaimer and limitations

---

FORMATTING REQUIREMENTS:
- Use markdown formatting with clear section headers (##, #
##)
- Use tables for financial data where appropriate
- Use bullet points and numbered lists for clarity
- Bold key metrics and verdicts
- Italicize interpretive commentary
- Include placeholders for charts: [CHART: Description]
- Keep language professional but accessible
- Explain technical concepts when first introduced
- Total length: aim for 8,000-12,000 words (comprehensive b
ut readable)

---
```

```
COMPLETE COMPANY PROFILE DATA:

{json.dumps(company_profile, indent=2, default=str)}

---

Generate the complete Bloomberg-style investment report no
w.
"""

    # ----------------------------------------------------------
--------------
    # Part 2: Call Gemini API
    # ----------------------------------------------------------
--------------

    print("\n📡 Calling Gemini API...")

    import requests

    url = f"https://generativelanguage.googleapis.com/v1bet
a/models/gemini-pro:generateContent?key={gemini_api_key}"

    payload = {
        "contents": [{"parts": [{"text": prompt}]}],
        "generationConfig": {
            "temperature": 0.3,  # Low temperature for fact
ual, consistent output
            "maxOutputTokens": 16000,  # Increased for comp
rehensive report
        }
    }

    try:
        response = requests.post(url, json=payload, timeout
=120)
        response.raise_for_status()
        result = response.json()
        report_text = result["candidates"][0]["content"]["p
arts"][0]["text"]

        print("✅ Gemini API call successful")

    except Exception as e:
        print(f"❌ Gemini API call failed: {e}")
        raise

    # -----------------------------------------------------
--------------
    # Part 3: Save report
    # -----------------------------------------------------
--------------

    with open('cache/investment_report.md', 'w', encoding
='utf-8') as f:
        f.write(report_text)

    print("\n✅ Report saved to cache/investment_report.m
d")
    print(f"   Report length: {len(report_text)} character
s")

    return report_text
```

12.2) Chart Generation (Optional Enhancement)

```
def generate_report_charts(cache: pd.DataFrame,
                             predictions: dict,
                             company_name: str):
    """
    Generate chart images to accompany the Gemini report.

    Charts:
    1. 2-year price history with regime shading
    2. Next week/month candlestick predictions
    3. Tier hierarchy bar chart
    4. Survival mode timeline
    5. Monte Carlo distribution
    """

    print("\n📊 Generating report charts...")

    import matplotlib.pyplot as plt
    import matplotlib.dates as mdates

    # Chart 1: Price history with regime shading
    fig, ax = plt.subplots(figsize=(14, 7))

    ax.plot(cache.index, cache['close'], linewidth=2, color
='black', label='Price')

    # Shade regimes
    regime_colors = {'bull': 'lightgreen', 'bear': 'lightco
ral', 'high_vol': 'lightyellow', 'low_vol': 'lightblue'}

    for regime, color in regime_colors.items():
        mask = cache['regime_label'] == regime
        ax.fill_between(cache.index, cache['close'].min(),
cache['close'].max(),
                            where=mask, alpha=0.2, color=colo
r, label=regime)

    ax.set_title(f"{company_name} - 2-Year Price History wi
th Regime Detection", fontsize=16, fontweight='bold')
    ax.set_xlabel('Date', fontsize=12)
    ax.set_ylabel('Price', fontsize=12)
    ax.legend(loc='best')
    ax.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('cache/chart_price_history.png', dpi=150)
    plt.close()
    print("  ✅ Price history chart saved")

    # Chart 2: Tier hierarchy weights (current)
    # ... implement additional charts ...

    print("✅ All charts generated")
```

12.3) PDF Conversion

```
def convert_report_to_pdf():
    """
    Convert markdown report to professional PDF using pando
c.

    Requires pandoc to be installed in Kaggle environment.
    """

    print("\n📄 Converting report to PDF...")

    import subprocess

    try:
        # Use pandoc to convert markdown to PDF
        subprocess.run([
            'pandoc',
            'cache/investment_report.md',
            '-o', 'cache/investment_report.pdf',
            '--pdf-engine=xelatex',
            '--toc',  # Include table of contents
            '--number-sections',  # Number sections
            '-V', 'geometry:margin=1in',  # 1-inch margins
        ], check=True)

        print("✅ PDF generated: cache/investment_report.pd
f")

    except subprocess.CalledProcessError:
        print("⚠ Pandoc not available. PDF conversion skipp
ed.")
        print("   You can convert manually with: pandoc inv
estment_report.md -o report.pdf")
    except FileNotFoundError:
        print("⚠ Pandoc not installed. Skipping PDF convers
ion.")
```

12.4) Complete Report Generation Pipeline

```
def run_report_generation_pipeline(company_profile: dict,
                                     cache: pd.DataFrame,
                                     gemini_api_key: str):
    """
    Complete Phase 4: Report generation pipeline.
    """

    print("\n" + "="*60)
    print("🚀 PHASE 4: REPORT GENERATION")
    print("="*60)

    # Step 1: Generate report with Gemini
    report_text = generate_bloomberg_report_with_gemini(
        company_profile=company_profile,
        gemini_api_key=gemini_api_key
    )

    # Step 2: Generate charts (optional)
    try:
        generate_report_charts(
            cache=cache,
            predictions=company_profile['predictions'],
            company_name=company_profile['company']['name']
        )
    except Exception as e:
        print(f"⚠ Chart generation failed: {e}")

    # Step 3: Convert to PDF (optional)
    try:
        convert_report_to_pdf()
    except Exception as e:
        print(f"⚠ PDF conversion failed: {e}")

    print("\n" + "="*60)
    print("✅ REPORT GENERATION COMPLETE")
    print("="*60)
    print("\nOutput files:")
    print("  - cache/investment_report.md (Markdown)")
    print("  - cache/investment_report.pdf (PDF, if pandoc 
available)")
    print("  - cache/chart_*.png (Charts)")

    return report_text

# Usage
report = run_report_generation_pipeline(
    company_profile=company_profile,
    cache=target_cache,
    gemini_api_key=GEMINI_API_KEY
)
```

---

12) Complete Operator 1 Execution Summary

Three-phase Kaggle workflow:

Notebook 1: Cache Building (~30 min)

1. User inputs company name
2. Resolve company via Eulerpool
3. Discover relationships via Gemini
4. Build 2-year cache (target + linked entities)
5. Compute survival flags + vanity
6. Export cache artifacts

Notebook 2: Temporal Analysis (~2-3 hours)

1. Load cache from Notebook 1
2. Detect regimes + structural breaks
3. Train 20+ mathematical modules
4. Run forward pass (day-by-day)
5. Burn-out process (10 iterations on 6 months)
6. Generate predictions (day/week/month/year)
7. Build complete company profile
8. Export profile JSON

Notebook 3: Report Generation (~15 min)

1. Load company profile from Notebook 2
2. Apply Technical Alpha protection (mask next-day intraday)
3. Send to Gemini API
4. Generate Bloomberg-style report
5. Export as PDF

Total pipeline time: ~3-4 hours per company

---

13) Kaggle notebook cell arrangement (recommended)

This project is complex and should be written as **multiple notebook cells**, each with a single clear purpose.

13.1 Principles for writing cells

- **One purpose per cell:** each cell should do one job (for example: "FMP client" or "Feature engineering").
- **Idempotent cells:** rerunning a cell should not corrupt outputs.
  - Prefer: "if cached artifact exists, load it" unless `FORCE_REBUILD = True`.
- **Hard boundaries:** keep external calls (Eulerpool/FMP/World Bank/Gemini) separated from feature engineering so you can debug data issues.

- **Optional sections are explicit:** optional cells should be clearly labeled and safe to skip.

13.2 Cell-by-cell layout (including optional cells)

1. **Intro + guarantees**
    - What will be produced.
    - What fails fast.
2. **Inputs**
    - Set `target_isin` and `fmp_symbol`.
3. **Load Kaggle secrets**
    - Read `EULERPOOL_API_KEY`, `FMP_API_KEY`, `GEMINI_API_KEY`.
4. **Global config**
    - Dates (last 2 years), budgets, timeouts, retries, `FORCE_REBUILD`.
5. **Shared HTTP utilities**
    - Retries, rate limiting, disk caching, request logging.
6. **Eulerpool client**
    - Profile, statements, peers, supply chain, executives.
7. **FMP client**
    - Quote verification, EOD OHLCV fetch, intraday helpers (optional).
8. **Verify identifiers (fail fast)**
    - Eulerpool profile for `target_isin`.
    - FMP quote for `fmp_symbol`.
    - Extract country from Eulerpool profile.
9. **World Bank macro module**
    - Fetch indicators and align with as-of logic.
10. **(Optional) Linked entities discovery (Gemini)**
    - Candidates only.
11. **(Optional) Linked entities resolution + budgets + queue**
    - Apply per-group and global caps.

- Write `cache/progress.json`.

12. **Build target daily OHLCV candles (FMP)**
    - The authoritative OHLCV series.
13. **Build target fundamentals daily table (Eulerpool + as-of alignment)**
14. **Feature engineering**
    - Returns, volatility, drawdown.
    - Liquidity/solvency/cash/valuation ratios.
    - Missingness + invalid-math flags.
15. **(Optional) Build linked caches in batches (group-by-group)**
    - Checkpoint after each entity/group.
16. **(Optional) Linked aggregates join**
    - Sector/industry/competitor aggregates into target table.
17. **Survival mode + hierarchy weights**
    - Compute survival flags and tier weights.
18. **Save cache artifacts**
    - Parquet outputs, `metadata.json`, request log.
19. **(Optional) Modeling / prediction (many cells)**
    - One cell per model. Wrap each in `try/except` and log failures.
20. **Report generation (Gemini)**
    - Generate markdown.
    - Include the required short LIMITATIONS section.
    - Convert to PDF (optional).

13.3 Suggested optional-cell marker convention

At the top of optional cells, add a clear guard:

```
RUN_OPTIONAL = {
    "linked_entities": True,
    "intraday": False,
    "heavy_models": True,
    "pdf": False,
}

if not RUN_OPTIONAL["linked_entities"]:
    print("Skipping linked entities")
else:
    # optional logic
    pass
```

This makes it easy to sell different report tiers without rewriting the notebook.