

課題①



server.py
Python

タイムアウトで起動しないのだが、原因究明して欲しい。
必要な情報があれば教えてください



Thought for 1m 44s >

3行サマリ

- 起動タイムアウトの第一容疑は“インポート時の重い初期化”（`matplotlib/figures` のトップレベル `import`、`OpenTelemetry/Jaeger` などの観測基盤初期化）です。 server
- 最短の切り分けは「重い `import` を遅延化」「観測系を環境変数で無効化」「起動プロファイルの採取」。下のコマンドをそのまま実行してください。
- 併せて最小パッチ（diff）を提示します。これで多くの環境で起動が即時化します（観測・Wolframは後段で再有効化）。

まずの結論（原因候補の優先度）

1. トップレベル `import` が重い
 - `from .figures import generate_all` が アプリ起動時に実行される設計。`matplotlib`/フォント探索/バックエンド選定で数十秒～分固まることがあります。 server
2. 観測基盤（`OpenTelemetry/Jaeger/OTLP`）の初期化
 - `setup_opentelemetry(...)` が 起動時に実行。環境変数 `OTLP_ENDPOINT` が不達だと接続試行でブロックする事例が多いです。 server
3. 巨大依存の起動時リンク（`NumPy/Pandas/MKL/OpenBLAS`）
 - 競合や乱数ライブラリのリンク遅延が出る環境があるため、**importタイムの可視化**で実測します。
4. `StaticFiles` マウントのパス解決
 - `BASE_DIR = Path(__file__).resolve().parents[2]` が想定外の階層を指し、巨大ディレクトリ走査を誘発している可能性（稀）。 server
5. `DB/Redis` クライアント
 - ここは `try/except` で失敗しても無効化されるため、ブロックの可能性は低いです（ログで“DB clients disabled”が出ていれば無関係）。
 server

即時の切り分け手順（そのままコピペでOK）

A. 起動プロファイルを採る（どこで止まるかを可視化）

```
# 1) import 時間の可視化 (どの import が重いか)
PYTHONPROFILEIMPORTTIME=1 python -X importtime -c "import backend.engine.server" 2> import_profile.log
sed -n '1,200p' import_profile.log

# 2) uvicorn 起動の詳細ログ (lifespan無効化で単純化)
MPLBACKEND=Agg \
OTLP_ENDPOINT="" \
OTEL_EXPORTER_OTLP_TRACES_TIMEOUT=1 \
uvicorn backend.engine.server:app --host 0.0.0.0 --port 4000 --log-level debug --lifespan off
```

B. 観測系を一时无効化（タイムアウト原因の切り分け）

```
# OpenTelemetry/Jaeger を無効化
export OTLP_ENDPOINT=""
export JAEGER_ENDPOINT=""
export OTEL_TRACES_EXPORTER=none
export OTEL_METRICS_EXPORTER=none
```

```
# matplotlib を X 非依存に固定
export MPLBACKEND=Agg

# 再起動
uvicorn backend.engine.server:app --host 0.0.0.0 --port 4000 --log-level debug --timeout-keep-alive 5 --lifespan off
```

C. ヘルス確認

```
curl -v http://localhost:4000/api/health
```

ここまでで **起動できる / できない** が分かります。ログ (`import_profile.log` と起動コンソール出力) を貼ってください。

最小パッチ (安全・可逆の遅延化)

目的: 起動時に重いものを一切ロードしない / 観測を環境変数で明示制御。

diff (`backend/engine/server.py` 対して)

```
- from .figures import generate_all # *追加
+ # NOTE: figures は重い依存 (matplotlib等) を含むため起動時 import を避ける
+ # 必要時に analyze() 内で遅延 import する

@@
- try:
-     from backend.observability.metrics import setup_observability_metrics
-     setup_observability_metrics(app, service_name="engine")
- except Exception as _e:
-     print(f"[observability][engine] metrics disabled: {_e}")
+ try:
+     import os
+     if os.getenv("CQOX_DISABLE_METRICS", "0") != "1":
+         from backend.observability.metrics import setup_observability_metrics
+         setup_observability_metrics(app, service_name="engine")
+     else:
+         print("[observability][engine] metrics disabled by CQOX_DISABLE_METRICS=1")
+ except Exception as _e:
+     print(f"[observability][engine] metrics disabled: {_e}")

@@
- try:
-     from backend.observability.tracing import setup_opentelemetry, instrument_fastapi
-     import os
-
-     otlp_endpoint = os.getenv("OTLP_ENDPOINT")
-     jaeger_endpoint = os.getenv("JAEGER_ENDPOINT", "localhost:6831")
-     enable_console = os.getenv("OTEL_CONSOLE", "false").lower() == "true"
-
-     setup_opentelemetry(
-         service_name="engine",
-         otlp_endpoint=otlp_endpoint,
-         jaeger_endpoint=jaeger_endpoint if not otlp_endpoint else None,
-         enable_console=enable_console
-     )
-     instrument_fastapi(app, service_name="engine")
-     print("[Tracing] OpenTelemetry enabled for Engine")
- except Exception as _e:
-     print(f"[observability][engine] tracing disabled: {_e}")
+ try:
+     import os
+     if os.getenv("CQOX_DISABLE_TRACING", "0") != "1":
+         from backend.observability.tracing import setup_opentelemetry, instrument_fastapi
+         otlp_endpoint = os.getenv("OTLP_ENDPOINT")
+         jaeger_endpoint = os.getenv("JAEGER_ENDPOINT", "localhost:6831")
+         enable_console = os.getenv("OTEL_CONSOLE", "false").lower() == "true"
+         setup_opentelemetry(
+             service_name="engine",
+             otlp_endpoint=otlp_endpoint,
```

```
+         jaeger_endpoint=jaeger_endpoint if not otlp_endpoint else None,
+         enable_console=enable_console
+     )
+     instrument_fastapi(app, service_name="engine")
+     print("[Tracing] OpenTelemetry enabled for Engine")
+ else:
+     print("[Tracing] disabled by CQOX_DISABLE_TRACING=1")
+ except Exception as _e:
+     print(f"[observability][engine] tracing disabled: {_e}")
```

さらに（任意だが推奨） — `analyze()` の図生成直前で遅延 import :

```
- from backend.engine.figures import generate_all
+ from backend.engine.figures import generate_all # analyze() の try 節の中に移動済みなら削除
```

これで **起動時は軽量、リクエスト時のみ重い依存をロード**します。

収集してほしい最小セット（貼ってください）

1. 直近の起動コマンド（`uvicorn ...`）
2. `import_profile.log` の先頭 200 行
3. 起動コンソールログ（`[observability]` や `matplotlib` の行が重要）
4. 環境変数の抜粋：

```
echo "OTLP_ENDPOINT=$OTLP_ENDPOINT JAEGER_ENDPOINT=$JAEGER_ENDPOINT CQOX_DISABLE_TRACING=$CQOX_DISABLE_TRACING
CQOX_DISABLE_METRICS=$CQOX_DISABLE_METRICS MPLBACKEND=$MPLBACKEND"
```

5. Python と主要依存のバージョン

```
python -V
python -c "import numpy, pandas, matplotlib; print(numpy.__version__, pandas.__version__)"
```

検証（pytest/メトリクス）

```
# diff 適用後の起動
CQOX_DISABLE_TRACING=1 CQOX_DISABLE_METRICS=1 MPLBACKEND=Agg \
uvicorn backend.engine.server:app --host 0.0.0.0 --port 4000 --log-level info --lifespan off

# 健康チェック
curl -s http://localhost:4000/api/health

# analyze のダミー実行（重い import が遅延で動くか）
curl -X POST http://localhost:4000/api/analyze/comprehensive \
-H 'content-type: application/json' \
-d '{"dataset_id":"demo","df_path":"data/demo/dataset.csv","mapping":{"y":"y","treatment":"treatment"}}' | head
```

What this does

- **Goal:** 起動タイムアウトを除去し、**起動→即** `/api/health` 応答を保証。
- **Inputs:** 既存コード（`backend/engine/server.py`） + 上記 diff / 環境変数。
- **Outputs/Side-effects:** 起動が軽量化。観測は `CQOX_DISABLE_*` で制御可能。重い図生成はリクエスト時のみロード。
- **Why this approach:** “**起動時に重い物を持たない**”のが Google/Meta/NASA 流の本番設計。障害切り分け（観測・描画・DB など）も**環境変数**だけでオンオフできる。
- **How to run/verify:** 上記コマンド群。`/api/health` が即応・`import_profile.log` で最大重みの import を特定。

Expert insight (Google/Meta/NASA級)

- 「**import** 時に何も仕事をしない」は本番 **API** の鉄則。**起動=軽い・リクエスト=重くてもOK**に分離すると、デプロイとヘルスチェックが劇的に安定します。

まずは上の切り分けコマンドの結果を貼ってください。

そのログに基づいて、**最小限の追バッチ**（必要なら `figures` 側の遅延化も）まで一気に詰めます。



frontend_new.log
File

engine_ultra_speed.log
File

frontend.log
File

engine_debug.log
File

parquet_pipeline.py
Python

1. ****WolframONE可視化**** - コード実装済みだが、統合エラー（構文問題）により無効化
2. ****反実仮想システム**** - 3系統実装済みだが、パフォーマンス理由で一時無効化
3. ****UI統合**** - Engine/Gateway/Frontend個別動作するが、E2E統合未完成
 - **✗** UTF-8エンコーディング問題未解決
 - **✗** E2E動作未証明
 - **✗** 可視化・結果未確認
 - ✗** 未解決:
 - API層のエラー（Engine → Gateway → Frontend）
 - データアップロード後の分析実行でInternal Server Error

ここもできなくて非常に困っている。

Thought for 2m 21s >



3行サマリ

- **E2E未完成の主因**は ①ポート/経路の不整合 と ②エンジン側マッピング例外 (**KeyError**)、加えて ③一部ブランチの**SyntaxError** です。UIは4000系に逃げ、Engineは8080で稼働=API到達できたりできなかったり。 frontend_new frontend engine_debug
- 500 の決定因は `received_discount_code` 列の欠落（UIから渡る mapping とDF列が不一致）。ここを**事前検証 + 自動変換 + 400**で明示に変えるのが最速復旧。 engine_debug
- Wolfram 統合は「無効化ログ」もある一方、**成功ログも出ており**、UI配線とUTF-8/フォント、出力ディレクトリ契約を詰めれば使えます。 engine_debug

事実ベースの切り分け

1. UI≠APIのポート不整合

- Vite が 4000→4001→...→**4006/4008** へポート移動。フロントが固定 `http://localhost:4000` を叩いていると失敗します。 frontend_new frontend
- Engine は **0.0.0.0:8080** で起動し `/api/health` 200。UI から 8080 に届いていない可能性が高い。 engine_debug

2. Engine 側の致命例外 (500)

- `KeyError: 'received_discount_code'` が `/api/analyze/comprehensive` で頻発。mapping.treatment と実DF列の齟齬が原因。 engine_debug

3. 別ツリーの SyntaxError

- `/backend/engine/server.py` の **行810 付近** `except` が壊れており、そのツリーではそもそも起動不能。複製ディレクトリ（「Copy 2」など）に壊れた版が残っている。 engine_ultra_speed

4. Wolfram 統合の状態

- ある起動では「WolframONE disabled...」だが、別ログでは **“WolframONE generation successful”** と出ており、コマンド経路と引数の形が合えば動くことを示しています。 engine_debug

5. Parquetパイプラインは設計妥当（契約オプション化/パケット化）

- ・ 契約あり→ゲート判定、契約なし→スキップで parquet+metadata を生成する構造は良い。これを **UIのアップロード導線** に正しく接続すればE2Eを閉じられる。 `parquet_pipeline`

直す順番（最短・確実）

(A) UI⇄API 接続を固定（最優先）

目的：ポート漂流やCORSに左右されず、常に `/api` が **Engine** に到達。

非目標：本番NginxやIstioの完全設計（あとでやる）。

1. Vite proxy を使う（開発時）

`vite.config.ts`（抜粋）

```
export default defineConfig({
  server: {
    port: 4006, // 固定
    proxy: { '/api': { target: 'http://localhost:8080', changeOrigin: true } }
  }
})
```

フロントからは `/api/...` の相対パスのみ使用。UIの `.env` は：

```
VITE_API_BASE=/api
```

2. Gateway 経由にする場合は、Gateway→Engine を 8080 に固定でプロキシ。どちらか一系統に統一（二重転送は封印）。

検証

```
# Engine
uvicorn backend.engine.server:app --host 0.0.0.0 --port 8080

# Frontend
pnpm dev

# 期待: http://localhost:4006 から /api/health → 200
```

（ログに `/api/health 200` が出ればクリア。 `engine_debug`）

(B) 500の根絶：mappingの健全化（Fail Fast 400）

目的：列不在で500を出さず、UIに修正指示を返す。

非目標：推測で“それっぽい列”を勝手に選ぶ（監査性が落ちる）。

最小パッチ（方針）

- ・ `analyze()` 入り口で **mapping** の存在検査と列存在検査。
- ・ 欠損なら **HTTP 400** と詳細メッセージ（利用可能な列の上位20件を同梱）。
- ・ treatment列が多値 or 文字列なら **安全な2値化ガード**（`{1,"1",true,"true","y","yes","treated"}→1` など）。
- ・ ここまで終えてから推定器に渡す。

理由：ログの `KeyError: 'received_discount_code'` は mapping と実データのズレ。ここで 400 を返せば UI 側で即修正できる。 `engine_debug`

検証

```
# 列が無い mapping を投げると 400 が返ること
curl -X POST http://localhost:8080/api/analyze/comprehensive \
  -H 'content-type: application/json' \
  -d '{"dataset_id":"demo","mapping":{"treatment":"received_discount_code"}}' -v

# -> HTTP/1.1 400 + メッセージ (利用可能列が出る)
```

(C) 「壊れた別ツリー」を排除

目的：起動元を1つに固定。

非目標：今は複数コピーの共存を許さない。

やること

```
# どの server.py を読んでいるか特定
python - <<'PY'
import backend,inspect
```

```
import backend.engine.server as s
print("server path:", inspect.getsourcefile(s))
PY

# “Copy 2” 等の壊れたツリーは rename か削除 (VCに残すならタグ)
mv "/home/hirokiionodera/cqox-complete (Copy 2)" "/home/hirokiionodera/cqox-complete_copy2_broken_$(date +%Y%m%d)"
```

※ SyntaxError が出ているツリーで起動していたログがあり、複製混在が不具合を招く。 ☐ engine_ultra_speed

(D) UTF-8/日本語の堅牢化 (アップロード/JSON/図)

目的: 文字化け・例外ゼロを既定化。

非目標: ロケール依存の暫定回避。

- 読込: `pd.read_csv(file, encoding="utf-8", encoding_errors="replace")` を第一候補に。
- JSON: FastAPI 既定を `ORJSONResponse` にし `ensure_ascii=False`。
- プロセス: `PYTHONUTF8=1 PYTHONIOENCODING=UTF-8` を service/env に。
- フォント: `Noto Sans CJK JP` をインストールし、matplotlib の `font.family` を Noto に固定。
- Vite: `<meta charset="utf-8">` を index.html で確認。

検証

```
# 日本語列ヘッダ/値入りCSVで analyze → 200
pytest -q tests/test_utf8.py -q
```

(E) Wolfram 統合: 契約の明示 & 失敗時のフォールバック

目的: 常に成功か、失敗でもAPIは200 (図リンクのみ欠落)。

非目標: Wolframエラーを500に伝播。

- `wolframscript -script` へ渡すJSON契約 (CSVパス、列名) を固定化。
- 出力はJSONで返す (作れたファイルの相対パス群)。
- 失敗時は catch して matplotlib フォールバック & 警告を結果JSONに付与。
- ログには `WolframONE generation successful` が出ており、配線はほぼ正。異常時だけ抑止すればOK。 ☐ engine_debug

検証

```
wolframscript -script wolfram/ate_animation.wl '{"csv":"data/demo/dataset.csv","y":"y","t":"treatment","time":"time"}' reports/demo
ls reports/demo/wolfram_ate_animation.gif
```

(F) Parquet パイプラインを UI に接続 (E2Eを閉じる)

目的: Upload→Packet (parquet+metadata) →Analyze を一本化。

- 既存 `ParquetPipeline.process_upload()` は 契約省略モードで parquet + metadata を生成できる (設計良)。これを Gatewayの `/api/upload/csv` → Engine /packets に繋げ、Analyze には packet の mapping をそのまま渡す。 ☐ parquet_pipeline

検証

```
# 1) UI からCSVをアップ
# 2) Engine が packets/<dataset_id>/data.parquet を生成
# 3) 「Analyze」押下 → mapping 付きで /api/analyze/... に到達 → 200 + 図/Decision
```

失敗時の観測・依存関係 (ノイズ抑制)

- DB系は `psycopg2` の **DependencyConflict** 警告が出るが、**DB clients disabled** でフェールセーフ化されており、今は無視可能。
☐ engine_debug
- OTelは有効だが、トレース到達に失敗してもアプリは上がっている。切り分け時は `CQOX_DISABLE_TRACING=1` で抑止。

具体コマンド (今日ここまで回す)

```
# 1) Engine 起動 (UTF-8/日本語/観測抑止で切り分け)
PYTHONUTF8=1 PYTHONIOENCODING=UTF-8 CQOX_DISABLE_TRACING=1 CQOX_DISABLE_METRICS=1 \
uvicorn backend.engine.server:app --host 0.0.0.0 --port 8080 --lifespan off --log-level info

# 2) Frontend (Vite proxy 固定)
pnpm dev # vite.config.ts の server.proxy を設定済みにする

# 3) 健康確認
curl -v http://localhost:4006/api/health

# 4) 500再現テスト (列欠落→400へ)
curl -X POST http://localhost:4006/api/analyze/comprehensive \
-H 'content-type: application/json' \
-d '{"dataset_id":"demo","mapping":{"treatment":"received_discount_code"}}' -v

# 5) 正常系 (正しい列名で)
curl -X POST http://localhost:4006/api/analyze/comprehensive \
-H 'content-type: application/json' \
-d '{"dataset_id":"demo","mapping":{"treatment":"treatment","y":"y"}}' | jq .
```

検証 (pytest/メトリクス)

- **tests/test_mapping_failfast.py** : 存在しない列→**HTTP 400**、メッセージに `available_columns` が含まれる
- **tests/test_utf8.py** : 日本語データ→**HTTP 200**、返却JSONが **UTF-8** (非ASCIIエスケープなし)
- **tests/test_proxy_e2e.py** : Vite proxy 経由で `/api/health` が **200**、`/api/analyze` が **200**

What this does

- **Goal** : E2Eが**確実に動く配線**と**500の撲滅**。
- **Inputs** : 既存 UI/Engine、logs、Parquet パイプライン。
- **Outputs/Side-effects** :
 - UI→常に `/api` (Vite proxy) 。
 - Engine→**mappingのFail-Fast化**で500→400に変換、UX/監査性↑。
 - UTF-8/日本語が**既定で安全**。
 - Wolframは**成功時は利用／失敗時は自動フォールバック**。
- **Run/Verify** : 上記 curl/pytest で即確認。
- **Why this approach** : **配線の確定と入力契約の厳密化**が最短復旧のレバー。重い最適化やIstioは後段でも、まず**「動くE2E」**を固定化するのが高ROI。

Expert insight (Google/Meta/NASAレベル)

- **“500は開発者の都合、400は利用者の都合”**。入力契約の破れは**必ず400で返す**と合意しておく、運用時のSLOもインシデントも激減します (根治的)。

参考ログ (根拠)

- Vite が 4006 を使用 (ポート漂流の事実) 。 `frontend_new`
- 別走で 4008 使用。 `frontend`
- Engine は 8080 で `/api/health` が 200、かつ `KeyError 'received_discount_code'` により 500。 `engine_debug`
- “Copy 2” ツリーで `server.py` に `SyntaxError`。 `engine_ultra_speed`
- Parquet パイプラインの契約/パケット設計。 `parquet_pipeline`

必要なら、上記 (A)～(E) を **最小diff** パッチとして私がまとめ直し、あなたのリポに当てた**自動確認済みアーカイブ**を作って返します。まずは (A)(B) の結果 (と 400/200 の応答ログ) を共有ください。

