Plan – Default-Aware Repayment Events (A) & Strategy Analysis Scripts (B)

1 Overall goal

We want to do two things:

A. **Fix instrumentation** so that for every run we have a *complete liability-level dataset*:

- 1 row per liability that matures during the simulation.
- `outcome` $\in$ {repaid, defaulted}.
- Trading stats (`buy_count`, `sell_count`, `net_cash_pnl`, `strategy`) for both repaid and defaulted liabilities.

B. **Add analysis scripts** that sit on top of the existing experiment output and answer:

- Why so many runs where dealers make no difference.
- How often traders actually use the dealer / VBT, and in what way.
- How successful the two main strategies are:
  (a) buying and holding to maturity, versus
  (b) buying then selling before maturity.
- How all this relates to parameters (κ, concentration, μ, `outside_mid_ratio`) and dealer capacity over time.

2 A. Default-Aware `repayment_events.csv`

2.1 A.1 Desired semantics

We want one CSV per run at:

`<run_dir>/out/repayment_events.csv`

with rows of the form:

```
run_id, regime, trader_id, liability_id, maturity_day, face_value,outcome, buy_count, sell_count
```

Where:

- `outcome` is now:
  - `"repaid"`: liability was fully settled at or before its maturity date.

  - `"defaulted"`: liability matured but was not settled when the simulation stopped (because someone defaulted).

- Only liabilities whose `maturity_day` is less than or equal to the final simulation day appear here.
- `strategy` is one of:
  - `"no_trade"` – `buy_count = sell_count = 0`.
  - `"hold_to_maturity"` – only buys (no sells) before maturity, ending holding.
  - `"sell_before"` – net position reduced/closed before maturity (one or more sells).
  - `"round_trip"` – at least one buy and one sell before maturity, ending flat.

We already have these `strategy` values, but currently all rows have `outcome = "repaid"`. The change is to add rows for defaulted liabilities as well.

2.2 A.2 Implementation sketch

A.2.1 Locate current repayment events builder

In the codebase, locate the code that generates `repayment_events.csv`. Likely places / clues:

- search for the string `"repayment_events.csv"`,
- search for strings `"no_trade"`, `"hold_to_maturity"`, `"sell_before"`, `"round_trip"`.

You should find something like a `build_repayment_events(...)` function that:

- iterates over completed liabilities,
- computes trading stats from `trades.csv`,
- writes out `repayment_events.csv`.

We will extend that function.

A.2.2 Build a complete liability map per run

For each run, build an internal map of all created payables and whether they were settled. We can either:

- reconstruct this from `events.jsonl`, or
- reuse an existing in-memory liability map from the engine if one already exists.

Example (Python-style sketch):

```
@dataclass
class LiabilityInfo:
    trader_id: str
    liability_id: str
    maturity_day: int
```

```
    face_value: int
    settled: bool = False
    settled_day: int | None = None


def build_liability_map(events_path: Path) -> tuple[dict[str, LiabilityInfo], int]:
    liabilities: dict[str, LiabilityInfo] = {}
    final_day = 0

    with open(events_path) as f:
        for line in f:
            e = json.loads(line)
            day = e.get("day", 0)
            final_day = max(final_day, day)

            kind = e["kind"]

            if kind == "PayableCreated":
                lid = e["payable_id"]
                liabilities[lid] = LiabilityInfo(
                    trader_id=e["debtor"],      # or equivalent field
                    liability_id=lid,
                    maturity_day=e["due_day"],
                    face_value=e["amount"],
                )

            elif kind == "PayableSettled":
                lid = e["pid"]                  # check field name
                info = liabilities.get(lid)
                if info is not None:
                    info.settled = True
                    info.settled_day = day

    return liabilities, final_day
```

The important part is to know, for each liability:

- the debtor (`trader_id`),
- maturity day,
- face value,
- whether it was settled by the end of the run.

A.2.3 Compute trading stats per liability

We already do this for repaid liabilities; the idea is to reuse that logic for all liabilities.

Given `trades.csv`:

```
def compute_trading_stats(trades_path: Path) -> dict[str, dict]:
    """
    Returns mapping: liability_id -> {
        'buy_count': int,
        'sell_count': int,
        'net_cash_pnl': float,
        'strategy': str,
    }
    """

    stats_by_liability = defaultdict(lambda: {
        "buy_count": 0,
        "sell_count": 0,
        "cash_in": 0.0,
        "cash_out": 0.0,
    })

    trades = pd.read_csv(trades_path)

    for _, row in trades.iterrows():
        # Only consider trader legs, not dealer-dealer, etc.
        trader_id = row["trader_id"]
        if not trader_id.startswith("H"):
            continue

        ticket_id = row["ticket_id"]   # e.g. TKT_PAY_...
        liability_id = ticket_id.replace("TKT_", "")

        entry = stats_by_liability[liability_id]

        if row["side"] == "BUY":
            entry["buy_count"] += 1
            entry["cash_out"] += row["price"]
        elif row["side"] == "SELL":
            entry["sell_count"] += 1
            entry["cash_in"] += row["price"]

    # finalize entries
    for lid, entry in stats_by_liability.items():
        entry["net_cash_pnl"] = entry["cash_in"] - entry["cash_out"]
        entry["strategy"] = classify_strategy(entry["buy_count"],
                                              entry["sell_count"])

    return stats_by_liility
```

Strategy classifier (keep aligned with current implementation):

```
def classify_strategy(buy_count: int, sell_count: int) -> str:
```

```
    if buy_count == 0 and sell_count == 0:
        return "no_trade"
    if buy_count > 0 and sell_count == 0:
        return "hold_to_maturity"
    if buy_count == 0 and sell_count > 0:
        # edge case (selling inherited claim): treat as sell_before
        return "sell_before"
    if buy_count > 0 and sell_count > 0:
        return "round_trip"
```

If a classifier already exists, we should reuse it; the key is to apply it to *all* liabilities, not just repaid ones.

A.2.4 Emit `repayment_events.csv` with defaulted rows

Using the liability map and trading stats, we produce the CSV:

```
def write_repayment_events(run_dir: Path, regime: str):
    events_path = run_dir / "out" / "events.jsonl"
    trades_path = run_dir / "out" / "trades.csv"
    out_path    = run_dir / "out" / "repayment_events.csv"

    liabilities, final_day = build_liability_map(events_path)
    trading_stats = compute_trading_stats(trades_path)

    rows = []

    for lid, info in liabilities.items():
        # Only consider liabilities that have actually matured
        if info.maturity_day > final_day:
            continue

        ts = trading_stats.get(lid, {
            "buy_count": 0,
            "sell_count": 0,
            "net_cash_pnl": 0.0,
            "strategy": "no_trade"
        })

        outcome = "repaid" if info.settled else "defaulted"

        rows.append({
            "run_id":        run_dir.name,   # e.g. active_k0.5_...
            "regime":        regime,         # "active" or "passive"
            "trader_id":     info.trader_id,
            "liability_id":  lid,
            "maturity_day":  info.maturity_day,
            "face_value":    info.face_value,
            "outcome":       outcome,
```

```
            "buy_count":     ts["buy_count"],
            "sell_count":    ts["sell_count"],
            "net_cash_pnl":  ts["net_cash_pnl"],
            "strategy":      ts["strategy"],
        })

    df = pd.DataFrame(rows)
    df.to_csv(out_path, index=False)
```

This can be called after each run completes, in the same harness that currently writes other output files.

A.2.5 Apply to passive runs

For passive runs:

- `trades.csv` will be absent or empty,
- `compute_trading_stats` will then produce all zeros with `strategy = "no_trade"`.

We can still emit `repayment_events.csv` for passive runs; all strategies will be `"no_trade"`, and `outcome` indicates which liabilities defaulted without a dealer.

This makes active vs passive comparisons symmetric.

2.3 A.3 Testing

1. Pick a run with known defaults (e.g. `delta_active > 0`).
    - Verify that `repayment_events.csv` contains both `outcome = "repaid"` and `outcome = "defaulted"`.
    - Check that the sum of face value for defaulted rows is consistent with what we expect from the default metrics (up to scaling).
2. Pick a run with no defaults (`delta_active = 0`, `phi_active = 1`).
    - Verify that all rows have `outcome = "repaid"`.
3. Spot-check that strategies for defaulted liabilities look reasonable (e.g., no-trade liabilities show `"no_trade"`, etc.).

3 B. Cross-Run Analysis and Aggregation Scripts

Once part A is implemented, we can add analysis scripts that operate purely on the output files.

We assume an experiment folder structure similar to:

```
out/experiments/detailed_instrumentation_sweep/
  aggregate/comparison.csv
  active/runs/<run_id>/out/...
  passive/runs/<run_id>/out/...
```

We propose two main scripts:

B.1. `analysis/build_strategy_outcomes.py`

B.2. `analysis/build_dealer_usage_summary.py`

They can be combined into a single module if preferred.

3.1 B.1 Script 1: `build_strategy_outcomes.py`

Purpose

For each run, summarize:

- how many liabilities used each strategy,
- how much face value each strategy covers,
- how often those liabilities were repaid vs defaulted.

This directly addresses: "To what extent are the strategies applied, and to what extent are they successful?"

Inputs

- `aggregate/comparison.csv` (experiment-level summary);
- all `repayment_events.csv` for active runs (and optionally passive).

Outputs

1. `aggregate/strategy_outcomes_by_run.csv`
2. `aggregate/strategy_outcomes_overall.csv`

B.1.1 `strategy_outcomes_by_run.csv` One row per run (typically the active side):

- basic run parameters: `run_id`, `kappa`, `concentration`, `mu`, `outside_mid_ratio`, `seed`, `face_value`, etc.;
- outcomes: `delta_passive`, `delta_active`, `trading_effect`, `trading_relief_ratio`;
- global counts: `total_liabilities`, `total_face_value`;
- for each strategy $s \in \{\text{no\_trade}, \text{hold\_to\_maturity}, \text{sell\_before}, \text{round\_trip}\}$:
  - `count_s`, `face_s`;
  - `default_count_s`, `default_face_s`;
  - `default_rate_s` = `default_face_s / face_s`.

Sketch:

```
def build_strategy_outcomes_by_run(experiment_root: Path):
    comp = pd.read_csv(experiment_root / "aggregate" / "comparison.csv")

    rows = []

    for _, row in comp.iterrows():
        run_id = row["active_run_id"]
        run_dir = experiment_root / "active" / "runs" / run_id

        rep_path = run_dir / "out" / "repayment_events.csv"
        if not rep_path.exists():
            continue

        rep = pd.read_csv(rep_path)

        total_face = rep["face_value"].sum()
        total_liab = len(rep)

        grouped = rep.groupby(["strategy", "outcome"])["face_value"] \
                     .sum().unstack(fill_value=0)

        def face_value_for(strategy, outcome):
            if strategy in grouped.index and outcome in grouped.columns:
                return grouped.loc[strategy, outcome]
            return 0.0

        strat_metrics = {}
        for strat in ["no_trade", "hold_to_maturity",
                      "sell_before", "round_trip"]:
            face_strat = rep.loc[rep["strategy"] == strat,
                                 "face_value"].sum()
            default_face = face_value_for(strat, "defaulted")

            strat_metrics[f"count_{strat}"] = \
                (rep["strategy"] == strat).sum()
            strat_metrics[f"face_{strat}"] = face_strat
            strat_metrics[f"default_face_{strat}"] = default_face
            strat_metrics[f"default_count_{strat}"] = len(
                rep[(rep["strategy"] == strat) &
                    (rep["outcome"] == "defaulted")]
            )
            strat_metrics[f"default_rate_{strat}"] = (
                default_face / face_strat if face_strat > 0 else 0.0
            )

        default_face_total = rep.loc[rep["outcome"] == "defaulted",
                                     "face_value"].sum()
```

```
        row_out = {
            "run_id": run_id,
            "kappa": row["kappa"],
            "concentration": row["concentration"],
            "mu": row["mu"],
            "outside_mid_ratio": row["outside_mid_ratio"],
            "seed": row["seed"],
            "face_value": row["face_value"],
            "delta_passive": row["delta_passive"],
            "delta_active": row["delta_active"],
            "trading_effect": row["trading_effect"],
            "trading_relief_ratio": row["trading_relief_ratio"],
            "total_liabilities": total_liab,
            "total_face_value": total_face,
            "default_face_total": default_face_total,
            "default_rate_total": (
                default_face_total / total_face if total_face > 0 else 0.0
            ),
        }
        row_out.update(strat_metrics)
        rows.append(row_out)

    out_df = pd.DataFrame(rows)
    out_df.to_csv(
        experiment_root / "aggregate" / "strategy_outcomes_by_run.csv",
        index=False
    )
```

B.1.2 `strategy_outcomes_overall.csv` Aggregate over runs, e.g. one row per combination of `(kappa, concentration, mu, outside_mid_ratio, strategy)`:

- `total_face`, `default_face`, `default_rate`;
- `runs_using_strategy`;
- optionally `mean_trading_effect`.

Sketch:

```
def build_strategy_outcomes_overall(experiment_root: Path):
    df = pd.read_csv(
        experiment_root / "aggregate" / "strategy_outcomes_by_run.csv"
    )

    rows = []

    for strat in ["no_trade", "hold_to_maturity",
                  "sell_before", "round_trip"]:
        group_cols = ["kappa", "concentration", "mu",
```

```
                "outside_mid_ratio"]

    for combo, group in df.groupby(group_cols):
        kappa, conc, mu, omr = combo

        face_total = group[f"face_{strat}"].sum()
        default_face = group[f"default_face_{strat}"].sum()

        rows.append({
            "strategy": strat,
            "kappa": kappa,
            "concentration": conc,
            "mu": mu,
            "outside_mid_ratio": omr,
            "total_face": face_total,
            "default_face": default_face,
            "default_rate": (
                default_face / face_total if face_total > 0 else 0.0
            ),
            "runs_using_strategy": (
                group[f"count_{strat}"] > 0
            ).sum(),
            "mean_trading_effect": group["trading_effect"].mean(),
        })

out = pd.DataFrame(rows)
out.to_csv(
    experiment_root / "aggregate" / "strategy_outcomes_overall.csv",
    index=False
)
```

This gives tables describing which strategies are used the most (by face value), which default the most, and how this depends on parameters.

3.2 B.2 Script 2: `build_dealer_usage_summary.py`

Purpose

Explain why so many runs show no effect from the dealer:

- Are dealers simply not used?
- Do they get emptied too quickly?
- Are trades happening, but mostly among already-safe traders?

Inputs

For active runs:

- `aggregate/comparison.csv`

- `trades.csv`
- `inventory_timeseries.csv`
- `system_state_timeseries.csv`
- (optionally) `repayment_events.csv` from part A.

Output

- `aggregate/dealer_usage_by_run.csv`

Metrics per run

From `trades.csv`:

- `dealer_trade_count` – number of trades,
- `trader_dealer_trade_count` – trades where a trader (H-entity) participates,
- `n_traders_using_dealer` – number of distinct traders who used the dealer,
- `total_face_traded` – total face value traded by traders,
- `total_cash_volume` – total cash volume traded by traders.

From `inventory_timeseries.csv` (aggregated per day):

- `dealer_active_fraction` – fraction of steps where any bucket has positive inventory,
- `dealer_empty_fraction` – fraction of steps where all buckets are at zero,
- `vbt_usage_fraction` – fraction of steps where any bucket hits VBT (`hit_vbt_this_step = True`).

From `system_state_timeseries.csv`:

- `mean_debt_to_money`,
- `final_debt_to_money`,
- `debt_shrink_rate` (from initial to final total face value).

Optionally, from `repayment_events.csv`:

- `frac_defaulted_that_traded` – share of defaulted face value where `used_dealer = True`,
- `frac_repaid_that_traded` – share of repaid face value where `used_dealer = True`.

Sketch

```
def build_dealer_usage_summary(experiment_root: Path):
    comp = pd.read_csv(
        experiment_root / "aggregate" / "comparison.csv"
    )
    rows = []

    for _, row in comp.iterrows():
        run_id = row["active_run_id"]
        run_dir = experiment_root / "active" / "runs" / run_id
        out_dir = run_dir / "out"

        trades = pd.read_csv(out_dir / "trades.csv")
        inv    = pd.read_csv(out_dir / "inventory_timeseries.csv")
        sys_ts = pd.read_csv(out_dir / "system_state_timeseries.csv")

        # trades metrics
        dealer_trade_count = len(trades)
        trader_trades = trades[trades["trader_id"].str.startswith("H")]
        trader_dealer_trade_count = len(trader_trades)
        n_traders_using_dealer = trader_trades["trader_id"].nunique()
        total_face_traded = trader_trades["face_value"].sum()
        total_cash_volume = trader_trades["price"].sum()

        # inventory metrics (aggregate by day)
        inv_by_day = inv.groupby("day").agg(
            any_positive=("dealer_inventory", lambda x: (x > 0).any()),
            any_zero    =("is_at_zero", lambda x: x.all()),
            any_vbt     =("hit_vbt_this_step", lambda x: x.any()),
        )

        n_steps = len(inv_by_day)
        dealer_active_fraction = inv_by_day["any_positive"].mean()
        dealer_empty_fraction  = inv_by_day["any_zero"].mean()
        vbt_usage_fraction     = inv_by_day["any_vbt"].mean()

        # system-wide debt metrics
        mean_debt_to_money  = sys_ts["debt_to_money"].mean()
        final_debt_to_money = sys_ts["debt_to_money"].iloc[-1]
        total_face0 = sys_ts["total_face_value"].iloc[0]
        total_faceT = sys_ts["total_face_value"].iloc[-1]
        debt_shrink_rate = (
            (total_face0 - total_faceT) / total_face0
            if total_face0 > 0 else 0.0
        )

        # optional: repayment_events-based metrics
```

```
rep_path = out_dir / "repayment_events.csv"
frac_defaulted_that_traded = None
frac_repaid_that_traded = None

if rep_path.exists():
    rep = pd.read_csv(rep_path)
    rep["used_dealer"] = (rep["buy_count"] + rep["sell_count"]) > 0

    def frac_used(outcome):
        sub = rep[rep["outcome"] == outcome]
        if sub.empty:
            return 0.0
        return (
            sub.loc[sub["used_dealer"], "face_value"].sum()
            / sub["face_value"].sum()
        )

    frac_defaulted_that_traded = frac_used("defaulted")
    frac_repaid_that_traded    = frac_used("repaid")

rows.append({
    "run_id": run_id,
    "kappa": row["kappa"],
    "concentration": row["concentration"],
    "mu": row["mu"],
    "outside_mid_ratio": row["outside_mid_ratio"],
    "delta_passive": row["delta_passive"],
    "delta_active": row["delta_active"],
    "trading_effect": row["trading_effect"],
    "trading_relief_ratio": row["trading_relief_ratio"],

    "dealer_trade_count": dealer_trade_count,
    "trader_dealer_trade_count": trader_dealer_trade_count,
    "n_traders_using_dealer": n_traders_using_dealer,
    "total_face_traded": total_face_traded,
    "total_cash_volume": total_cash_volume,

    "dealer_active_fraction": dealer_active_fraction,
    "dealer_empty_fraction": dealer_empty_fraction,
    "vbt_usage_fraction": vbt_usage_fraction,

    "mean_debt_to_money": mean_debt_to_money,
    "final_debt_to_money": final_debt_to_money,
    "debt_shrink_rate": debt_shrink_rate,

    "frac_defaulted_that_traded": frac_defaulted_that_traded,
    "frac_repaid_that_traded": frac_repaid_that_traded,
})
```

```
out_df = pd.DataFrame(rows)
out_df.to_csv(
    experiment_root / "aggregate" / "dealer_usage_by_run.csv",
    index=False
)
```

Once this is in place, we can:

- filter runs with approximately zero `trading_effect` and check:
  - whether `trader_dealer_trade_count` is basically zero (dealer not used),
  - whether `dealer_active_fraction` is tiny (dealer empties quickly),
  - whether most defaulted liabilities come from traders who never used the dealer.

This directly addresses the question of why so many cases show no effect from the dealer.