ChatGPT

Project Scope and Goals

- **Comprehensive A/B Testing Curriculum:** The repository is a full walkthrough of an experimentation playbook, using a realistic **subscription onboarding A/B test** as the running example. It simulates 100,000 users split into Control (A) vs. Treatment (B), with rich data (pre-experiment engagement, revenue, sessions; user segments like channel/device; conversion and revenue outcomes; plus extra metrics like sessions, pages, retention, NPS).
- **End-to-End Experiment Workflow:** The content covers the lifecycle from **experiment design (sample size/power)** to **analysis (statistical tests, variance reduction, etc.)**, to **decision-making (interpretation and launch decisions)**. It's organized into *Levels 1–4* (Fundamentals → Advanced) and demonstrates each technique on the simulated data with detailed reasoning and formulas.
- **Key Techniques Covered:** It includes classical methods (e.g. power analysis, **z-test for proportions**, **Welch's t-test** for means, **Mann–Whitney U** for non-parametric comparisons, **bootstrap confidence intervals**, **SRM checks** for randomization sanity) and more advanced ones (**Bayesian beta-binomial** analysis, **multiple testing corrections** like Bonferroni/BH, **CUPED** and **CUPAC** for variance reduction, **sequential testing** with O'Brien-Fleming boundaries, **heterogeneous treatment effect** modeling with an X-learner, **quantile treatment effects**, the **delta method** for ratio metrics, **winsorization** for outliers). Each technique is contextualized in the scenario (e.g. interpreting conversion lift, revenue impact) for practical understanding.
- **Learning & Interview Prep Focus:** The primary goal is **to learn and practice modern experimentation techniques in a realistic DS workflow** – from designing tests to analyzing and drawing business decisions. The content emphasizes not just how to run the numbers, but also the "talk track" behind each method (e.g. explaining what a p-value *really* means, or why variance reduction helps). This prepares the user for analytics interviews by providing both implementation patterns and the reasoning you'd need to articulate.
- **Repository Improvement Goals:** Now, the aim is to **modularize and productionize** this repo. Currently it's largely one big script (`ab_testing_framework.py`), which is great for learning but not ideal for reuse. The goal is to refactor it into a clean **Python package** with logical modules (power analysis, testing methods, variance reduction, etc.), each with reusable functions and possibly classes. This modular structure will make the content easier to maintain, extend, and test. It also aligns with production best practices, showing interviewers that you can write well-organized, production-quality code rather than monolithic scripts.
- **Hands-on Practice and Extendability:** Another goal is to enrich the repo with **"labs" or exercises** – interactive notebooks or prompts where you can practice each concept (e.g. fill in the code to run a CUPED adjustment, interpret an SRM result, etc.) and verify your answers. In addition, we plan to add **unit tests** for core functions and set up CI (e.g. using `pytest` and a linter like `ruff`) to ensure the code is correct and stays that way. This both instills good software engineering habits and provides confidence in the correctness of our analytical methods.
- **Up-to-Date Techniques & Insights:** We want to update the curriculum to reflect **the latest best practices in experimentation (2024–2025)**. This means incorporating recent developments and nuanced topics that modern data science teams emphasize: e.g. **guardrail metrics** and multiple-metric decision frameworks, handling **novelty effects** and seasonal trends, dealing with **interference** in experiments (network effects) and solutions like **cluster randomization**, formal analysis of **non-compliance** (e.g. using **Instrumental Variables** for encouragement designs), and

other pitfalls and diagnostics that go beyond the current content. By adding these, the repo will not only cover the "classic" material but also demonstrate awareness of cutting-edge experimentation research – helping differentiate the user as a sophisticated experimentation data scientist.

Current State of the Repository (Strengths & Gaps)

What's Strong: The repo in its current form is a **comprehensive, well-explained tutorial** on A/B testing. It's impressive in that it doesn't just show code outputs – it provides context, reasoning, and interpretation for each result. For example, it prints out interpretations of p-values, explains why we use Welch's t-test instead of Student's, and walks through the math of power analysis. This is great for learning. The simulated dataset is quite realistic (with meaningful correlations and segments), so techniques like CUPED/CUPAC and HTE have data to work with that mirror real-world behavior. The **breadth of topics** covered (from basic hypothesis tests up through advanced methods like X-learners and sequential tests) means the repository already hits many core skills expected of a data scientist working on experiments. In short, the content is pedagogically strong and technically rigorous, making it a solid foundation.

Monolithic Structure: The biggest weakness is **maintainability and modularity**. All the logic lives in one large script/notebook. This makes it hard to reuse specific pieces in new analyses – e.g. if I wanted to apply just the SRM check on a different dataset, I'd have to extract that code manually. It also makes testing in isolation difficult (no unit tests currently). In production or even in interviews, you'd want to show that you can package your methods cleanly. Right now, the repo lacks a clean separation of concerns (e.g. a function for computing a confidence interval vs. printing formatted text). Refactoring into a package with well-defined modules (and functions like `calculate_power()` or `run_z_test()`) will greatly improve this. It will reduce repetition (if future expansion happens) and allow adding automated tests.

Missing Real-World Edge Cases: While the simulated experiment is realistic, the curriculum currently glosses over some **practical challenges and pitfalls** that experimentation teams worry about in production:

- **Metric Pitfalls & Validation:** The repo assumes metrics are correctly implemented and stable. In reality, experimenters often do **"sanity checks" on metrics**, including running A/A tests or verifying that pre-experiment metrics are balanced between groups. For example, one might check that engagement or revenue *before* the experiment is statistically equal in A vs B (to catch any unintended segmentation). The current SRM check covers unequal sample counts, but not these content validity checks. There's no mention of running an **A/A test** or checking tracking integrity. Adding some metric validation (like ensuring no metric moves in an A/A scenario) could be valuable [1] [2].

- **Interference and Clustering:** The scenario assumes each user's outcome is independent. In modern social or networked products, this often isn't true – one user's treatment can affect another (interference). For example, on a social network, if only some users see a feature, their interactions with friends in the other group can "spill over," diluting measured effects. The repo doesn't cover this interference issue or strategies like **cluster randomization** to handle it. In practice, if interference is present and not addressed, it can bias results or wash out real effects [3]. (E.g., Meta has discussed how a test on a calling feature saw the effect disappear at scale because calls involved both groups, causing control users to partly experience the treatment through their interactions [3].) We should at least mention this limitation and how to detect or mitigate it (perhaps via designing clusters of users and randomizing at cluster level to contain the effect [4]). Not covering interference is a gap because interviewers might ask “What if users in different groups interact?” to test the candidate’s depth.

- **Novelty and Duration Effects:** The current analysis treats the experiment as a static snapshot. In reality, *time* matters – new features often show a **novelty effect** (a temporary surge or dip in metrics due to the “newness”, not the true long-term impact) [5]. The repo doesn’t demonstrate analyzing metrics over time or checking if the effect persists. For example, an experiment might look great in week 1 but then the lift fades in week 3 as users acclimate (or even reverses if users get annoyed). Modern best practices include looking at **time-series of metrics or post-launch holdout periods** to distinguish novelty vs. sustained impact [6] [7]. Adding a section on analyzing **metric trends over time** or discussing novelty effects (and how not to be misled by them) would strengthen the curriculum. Otherwise, a user of this repo might think a significant test result is always good to ship, without realizing it could be a short-term blip.

- **Logging & Assignment Issues:** The repo assumes perfect random assignment (50/50) and that the assignment is implemented correctly (which is why SRM passes). In real experiments, many things can go wrong: users could be improperly split (e.g. hashing by a non-stable ID leading to duplicates or changes), or logging issues could cause some events not to be tracked for one group. These can manifest as an SRM or other anomalies. While the repo can’t simulate every logging bug, it currently doesn’t mention this class of problems. A note or exercise on **debugging SRM causes** (bot traffic, cookie churn, etc.) would be valuable [8]. Also, highlighting the importance of using persistent user identifiers and consistent logging for assignment would show production awareness.

- **Multi-metric Decision Making:** The content does show multiple metrics and how to correct p-values when looking at many. However, it doesn’t explicitly discuss **guardrail metrics** or how to combine results into an overall decision beyond a simple decision matrix. In practice, experiments have **primary metrics** (what you’re trying to improve) and **guardrail (counter) metrics** (what you *don’t* want to hurt – e.g. you might test a UI change aimed at boosting engagement, but you use conversion or retention as guardrails to ensure you’re not tanking those) [2]. The repository’s decision framework is a basic significance/positive matrix. Modern approaches (like Spotify’s) formalize this: you treat guardrails with non-inferiority tests and possibly don’t punish them in multiple-testing corrections [9]. The repo doesn’t cover non-inferiority or how to handle conflicting metric movements in a principled way. This is a possible risk if a learner oversimplifies decisions. We should integrate guardrail thinking – at least qualitatively – so the user knows that “significant improvement on metric X means ship” is not the whole story if metric Y suffered.

- **Advanced Causal Inference Context:** The repo introduces heterogeneity via X-learners, which is fantastic. One thing it doesn’t do is connect the **ITT vs. per-protocol** analysis to a more formal causal framework. In an encouragement design (where not everyone in B got the treatment, as we simulated with 90% trigger), the per-protocol lift is actually the **Complier Average Treatment Effect (CATE or LATE)**. There’s a whole body of causal inference on using **instrumental variables (IV)** to estimate the true effect on those who were treated vs. not (taking treatment assignment as an instrument for actual exposure) [10]. The repo calculated the difference, but without explaining it in IV terms or the assumptions (no defiers, exclusion, etc.). While implementing an IV regression might be beyond scope, discussing this link would show a deeper understanding. It’s somewhat risky that a user might report the per-protocol lift without understanding that it’s a biased estimator if non-compliers differ – or that there's a formal formula to derive it (Effect_PP ≈ Effect_ITT / take-up). Adding a brief segment on **encouragement design** and IV (with the concept of always-takers/never-takers) [11] [12] can solidify this part.

- **Other Potential Gaps:** The current repo doesn't explicitly mention **clustered standard errors or segmentation analysis** (aside from X-learner). In cases where randomization is by cluster (e.g. testing a change on a store-by-store basis, or a classroom by classroom basis), one must use cluster-robust SEs or treat the unit of analysis as the cluster. While our simulation is user-level, noting this in the documentation would be good. Also, concepts like **bandit algorithms** or continuous experimentation aren't covered – which is fine, as they're somewhat separate, but it's worth the author knowing they exist (maybe an aside: A/B testing vs. multi-armed bandits). Lastly, **novelty** and **seasonality** we already mentioned; we might also note **"Twyman's law"** – if an experiment's effect is *too* large or weird, double-check instrumentation (e.g. a bug could fake a huge lift). The repo does caution about p-value misinterpretation, which is good. We'll want to continue that spirit of honesty by exposing these real-world pitfalls, not hiding them.

In summary, the repo is a strong learning tool but needs modular structure and inclusion of these real-world complexities to be a truly up-to-date resource. Addressing these will reduce the risk of a learner picking up bad habits or blind spots (like shipping a test with a novelty spike or failing to notice interference).

Modularization Plan: Package Structure and API

We will refactor the code into a **Python package** (e.g. named `ab_testing`) under `src/ab_testing/`, with each major topic in its own module. This improves clarity and reusability. Below is a proposed folder structure and the responsibilities of each module:

```
src/ab_testing/
├── __init__.py           # Make it a package (could define high-level imports
here)
├── power.py              # Sample size and power analysis functions
├── srm.py                # Sanity checks like SRM (sample ratio mismatch) and
randomization balance
├── frequentist.py        # Frequentist test functions (z-test, t-test, chi-
square, etc.)
├── bayesian.py           # Bayesian A/B testing functions (e.g., beta-binomial
posterior)
├── variance_reduction.py # Variance reduction utilities (CUPED, CUPAC
adjustments)
├── sequential.py         # Sequential test boundaries and peek adjustments
├── multiple_testing.py   # Multiple comparisons corrections (FWER, FDR methods)
├── ratio_metrics.py      # Delta method for ratio metrics and related
calculations
├── noncompliance.py      # ITT vs. per-protocol analysis, and optional IV/LATE
calculation
├── hte.py                # Heterogeneous Treatment Effects (X-learner, segment
effects)
├── diagnostics.py        # Experiment diagnostics (guardrail checks, novelty
analysis hooks)
├── reporting.py          # Utility for formatting results, decision logic, and
summaries
└── simulation.py         # Data generator(s) for synthetic experiments
```

Module Responsibilities and Key Functions:

- `power.py`**:** Functions for power and sample size calculations. For example: `required_samples_binary(p_baseline, mde, alpha=0.05, power=0.8, two_tailed=True)` to return the per-group sample size for a binary metric (using Cohen's h or normal approximation)[13], and `required_samples_continuous(mu, sigma, mde, alpha=0.05, power=0.8, two_tailed=True)` for continuous metrics (using Cohen's d). We can wrap `statsmodels.stats.power` or implement formulas directly. The module can also include helpers like `cohens_h(p1, p2)` and `cohens_d(m1, m2, s1, s2)` if needed. This keeps all power analysis logic in one place.

- `srm.py`**:** This module will have functions to detect sample ratio mismatch and possibly other randomization sanity checks. For example, `check_srm(n_control, n_treatment, alpha=0.001)` could perform a chi-square test comparing observed vs. expected allocation[14] and return a boolean or p-value. Another function might be `baseline_balance_test(df, cols, treat_col)` which takes the pre-experiment features (like `pre_engagement`, `pre_revenue`) and ensures there's no significant difference across variant groups for each (e.g. a series of t-tests or KS tests). This isn't in the current repo, but adding it underscores good practice (it could flag if randomization was stratified or if something odd happened). Keeping these in `srm.py` (which might be renamed to something like `randomization.py` or `sanity.py`) isolates all pre-checks.

- `frequentist.py`**:** All frequentist hypothesis tests go here. This includes: `z_test_proportions(x_control, n_control, x_treatment, n_treatment, two_tailed=True)` implementing a two-proportion z-test (returning test statistic, p-value, and perhaps confidence interval). `t_test(measurements_control, measurements_treatment, two_tailed=True, equal_var=False)` wrapping Welch's t-test (using SciPy) for continuous metrics. We could have `chi_square_test(observed_counts, expected_props)` for general chi-square, though SRM could call that. Also, include `mann_whitney_u(control_values, treatment_values)` returning the U statistic, p-value, and maybe a rank-biserial effect size as in the current code. By centralizing these, we avoid duplicating math and can reuse them in different contexts (e.g. a lab could call `frequentist.z_test_proportions` easily). This module basically becomes a mini library of stats tests similar to `scipy.stats`, but tailored to experiments (with sensible defaults and print interpretations possibly separate).

- `bayesian.py`**:** Functions for Bayesian A/B testing. The current repo uses a Beta-Binomial for conversion rates. We can implement `beta_binomial_posteriors(x_control, n_control, x_treatment, n_treatment, prior=(1,1))` which returns the posterior alpha & beta for control and treatment. Then `bayesian_prob_B_better(alphaA, betaA, alphaB, betaB,` `samples=10000)` to simulate and compute P(B>A) and expected loss, etc. Another function could directly compute the closed-form probability that B > A for Beta-Binomial (there are known integrals, or just use simulation as in code). If we extend Bayesian methods, we might include a function for continuous metrics using a t-distribution or normal-inverse-gamma prior – but that might be overkill. We will keep this module minimal (because heavy Bayesian tools would require PyMC3/4 or similar, which we want to keep optional). Ensuring we document that Bayesian results are **probabilistic statements** is key. The module can also include a small function to format a credible interval from posterior samples.

- `variance_reduction.py`**:** This houses both CUPED and CUPAC logic. For CUPED: a function `cuped_adjust(Y, X)` that returns the adjusted outcome array given outcome `Y` and pre-experiment covariate `X` (computing θ = Cov(Y,X)/Var(X) [15]). Possibly a variant `cuped_diff(control_Y, treatment_Y, control_X, treatment_X)` that returns the adjusted difference in means and its variance. For CUPAC: we can have `cupac_adjust(Y, X_features)` that takes outcome and a matrix of features, trains a model (perhaps allow passing in a pre-trained model or specify model type), returns adjusted Y. We'd likely use `sklearn.ensemble.GradientBoostingRegressor` or similar as in current code, with cross-validation to get out-of-fold predictions [16]. We might create a small class `CupacModel` that fits the model on control data, predicts for all, and then adjusts Y – but a simple function is fine. The module should also include utility to compute variance reduction achieved (e.g. return the ratio of variances). By making this a module, the user could easily plug in their own data with pre-experiment metrics and see how much variance reduction they'd get – a practical skill. Documentation can cite that variance reduction can save a lot of sample size (DoorDash saw ~40% test duration reduction with CUPAC in some cases [17]).

- `sequential.py`**:** Functions to support sequential (peek) testing. One function `obrien_fleming_boundary(alpha=0.05, look=1, total_looks=1)` can return the Z-score cutoff for a given interim look using the O'Brien-Fleming approach [3]. For example, if total looks = 5 and we're at look 3 (information fraction = 0.6), it would return a higher Z threshold ~ perhaps 2.6 instead of 1.96. We might also include `pocock_boundary()` if we want a constant boundary alternative. Another function `sequential_decision(z_stats, current_look, total_looks, method='obrien-fleming')` could wrap the logic: it takes the current test statistic and determines whether to stop early or continue. This module should emphasize that sequential methods control Type I error across multiple peeks, preventing the inflated false positive risk from naive peeking (which can be ~14% for 5 looks at 5% alpha) [18] [19]. With these utilities, an advanced learner could simulate an experiment monitoring plan. We'll keep dependencies minimal (just SciPy for the normal CDF/inverse).

- `multiple_testing.py`**:** Contains methods for adjusting p-values or significance thresholds when multiple metrics or variants are tested. We can expose a function `bonferroni_adjust(p_values, alpha=0.05)` that returns a cutoff or adjusted p-value list, and `bh_adjust(p_values, alpha=0.05)` that returns the Benjamini-Hochberg critical values and which hypotheses are significant [20] [21]. We might simply wrap `statsmodels.stats.multitest.multipletests` for convenience, but it's good to show the basic procedure for BH (sorting p's, etc.) for transparency. The current content demonstrates the difference between raw, Bonferroni, and BH in terms of how many metrics declare significance – we can retain that as either part of the notebook or as a function that prints a nice summary table. The module can also mention other techniques (Holm, Hochberg, etc.) but BH is fine. Additionally, since the *Spotify "risk-aware" approach* [9] *suggests not* adjusting alpha for guardrails*, we can document that guardrail metrics (non-inferiority tests) might be treated differently – but that's more a conceptual note; implementation-wise, non-inferiority uses one-tailed tests. We won't delve into bespoke decision engines in code, but we'll note how to interpret corrections properly in context (e.g. avoid too-conservative Bonferroni if you have many exploratory metrics, lean on BH for discovery).

- `ratio_metrics.py`**:** A focused module for ratio metric analysis using the delta method. This will include a function like `delta_method_diff(num_control, den_control, num_treatment, den_treatment)` which returns the estimated difference in ratios (e.g. conversion = clicks/impressions) and a standard error & confidence interval [14]. Internally it will compute the variance of a ratio estimator: given E = N/D, we use the delta method formula Var(E) ≈ (1/μ_D^2)*Var(N) - 2*(μ_N/μ_D^3)*Cov(N,D) + (μ_N^2/μ_D^4)*Var(D) [22]. The code in the script already had a function for revenue per user. We'll generalize that. This module can also include simpler helpers like `pooled_mean(values)` or `cov(x,y)` if needed, or even a check that the delta method's normal approximation is reasonable (for large samples it is). We'll ensure to mention that ratio metrics (like ARPU, click-through rate) should ideally be analyzed via delta method or other proper techniques rather than treating them as simple means, because "ratio of means != mean of ratios" and ignoring that can give mis-estimated confidence intervals [14]. By having a dedicated function, users can plug any numerator/denominator arrays and get a result.

- `noncompliance.py`**:** This module covers analyses when not everyone in the treatment group actually received the treatment (or some in control got it – though in our sim, control group "triggered" just means they saw the control experience). We will provide: `itt_effect(metric_control, metric_treatment)` – basically just difference in means (already trivial, but for completeness), and `per_protocol_effect(metric_control_triggered, metric_treatment_triggered)` – difference in means among those who actually received the treatment vs. control (like we did filtering on `triggered==1`). More importantly, we add `calculate_cace(ITT_effect, treat_take_rate, control_take_rate=0)` which computes the **Complier Average Causal Effect (CACE)** using the formula CACE ≈ ITT / (take_rate_treatment - take_rate_control) [10]. In our scenario, control_take_rate is 0 (no one in control can get the new onboarding), and take_rate_treatment is 0.90 (90% triggered), so CACE = ITT / 0.9 – essentially what the per-protocol measured. This function formalizes that calculation and can be documented with assumptions (monotonicity, exclusion, etc.). We can also include an example of using an instrumental variable regression: e.g. `estimate_cace_via_iv(data, outcome_col, treat_flag_col, actual_exposure_col)` which uses `statsmodels` 2SLS (if we allow that dependency) to illustrate getting the same result analytically. However, a simpler approach is fine: just note that if some treatment users don't get the treatment, the ITT underestimates the *actual* effect on those who comply. The module will also outline the interpretation differences: ITT is the conservative, unbiased estimator for policy (what if we roll out to everyone with the same uptake?); CACE/LATE is the effect on those who actually got the feature (useful to diagnose implementation issues or to know the upside if we could get everyone to use it). We should caution that if take-up is very low, CACE might be large but based on a small group (no free lunch: standard errors will be bigger, as Spotify notes – power comes from compliers only [23]).

- `hte.py`**:** Heterogeneous Treatment Effects analysis. We have a custom X-learner in the script – we can turn that into a function `estimate_cate_x_learner(X, y, treatment, outcome_model=GradientBoostingRegressor, propensity=None)` which returns individual CATE estimates and perhaps summary stats (ATE, ATT, etc.). We'll likely use `sklearn.ensemble.GradientBoostingRegressor` by default as in the script (or allow any regressor class to be passed). This function would: fit μ0(x) on control data, μ1(x) on treatment data; compute pseudo outcomes τ0 and τ1; fit τ models; then combine into CATE estimates (the standard X-learner algorithm) [24] [25]. We can also provide a simpler function `segment_effects(data, segment_col, metric_col, treat_col)` to compute differences in subgroups (like conversion lift for each channel or device) with statistical tests. That gives a baseline for heterogeneity (sometimes a t-test per segment or a forest plot of effects). The module could mention advanced alternatives: e.g. **causal forests** (Athey's GRF) or libraries like Microsoft's `econml` which have an XLearner implementation. We might not integrate those due to dependencies, but we can reference them. By providing an X-learner implementation, we show we understand the method, and it's a great interview talking point ("I implemented a meta-learner to estimate individual treatment effects and found segments with higher lift"). The module can output things like the distribution of CATE, and allow the user to identify users or segments with the highest predicted treatment effect. We should include caveats: HTE detection risks multiple comparisons (you could find spurious segments if you hunt without hypothesis), and that one should ideally adjust for that or use it for hypothesis generation. We can tie this to business: e.g. personalization – if a certain segment has a much higher positive CATE and others are neutral or negative, maybe target that segment for rollout and not others.

- `diagnostics.py`**:** This module acts as a collection of experiment "health" checks and perhaps time-series analyses. For example, `check_guardrails(metrics: Dict[str, float], thresholds: Dict[str, float])` could evaluate a set of guardrail metrics against acceptable degradation thresholds (e.g. "significance in negative direction beyond -2% is a failure"). Or a function `analyze_novelty(metric_time_series_control, metric_time_series_treatment)` that could, for instance, fit a simple trend or just compute the treatment effect over time (day by day conversion rates) and see if it's decaying. In absence of real-time data, we might simulate an effect that wears off to demonstrate how one would detect it (looking for a downward slope in the lift). We might also include a `check_outliers(metric_values)` to flag if a metric distribution is extremely skewed (though we do winsorization separately). Essentially, `diagnostics.py` is a place for any miscellaneous checks that don't fit elsewhere: ensuring data quality and analyzing whether assumptions hold. In a production experiment analysis pipeline, this might include things like: test for normality if needed, test for variance equality (for deciding t-test vs Welch), plotting metric distributions, etc. For our purposes, a key diagnostic is **trend analysis** for novelty and **guardrail monitoring**. We can leverage the guardrail concept: define some metrics as guardrails in our sim (say 7-day retention or NPS) and illustrate checking "did any guardrail metric significantly worsen?". If yes, the decision might be "Hold despite primary metric improvement." Having a utility to perform a **non-inferiority test** could live here too (which is basically a one-sided test that metric >= -δ difference with high confidence). We can implement a simple version: e.g. `non_inferiority_test(diff, se, delta, alpha)` that checks if the lower bound of the CI is above -delta. That would formalize guardrail acceptance criteria.

- `reporting.py`**:** This module focuses on presentation and aggregation of results. It might include functions to format outputs for easy reading or to combine metrics. For instance, `format_effect(size, ci_low, ci_high, p_value)` that returns a nicely formatted string of the effect ± CI and stars for significance – useful for quickly generating report lines. Or `decision_recommendation(primary_metric_result, guardrail_results)` that implements the decision matrix more formally: e.g. returns "SHIP" if primary_metric is significantly positive and no guardrails significantly negative, "DON'T SHIP" if primary is significantly negative or guardrail fails, etc. Potentially, a `build_report(dataframe, metrics_list)` that runs all necessary tests on a set of metrics and produces a summary (this could tie everything together, effectively an automated analysis). Since the user likely will still do the final interpretation, this module mainly helps avoid repetitive manual calculations when writing the report. We might also add visualization helpers here (though keeping things textual is fine). If we wanted, we could output markdown or JSON summarizing the experiment – demonstrating some productization. However, given time, focusing on textual summary is sufficient. We will ensure the **decision logic aligns with best practices**: for example, incorporate the idea from the Spotify paper that all metrics need to be considered together [26]. We might not build the full decision engine, but we'll mention that a real system would categorize metrics (primary, guardrail, etc.) and have logic for each.

- `simulation.py`**:** This will contain the data generation code (the `DataGenerator` class from the script, possibly renamed to just functions). We can keep `subscription_experiment(n=100000, seed=42)` that returns the DataFrame as before. We might also add simpler simulation functions for specific scenarios if needed (like an A/A generator, or a cluster-interference generator for an exercise). But the main one is the subscription onboarding scenario which already includes interesting features (heterogeneity, pre metrics, etc.). By isolating simulation, we separate the "toy data creation" from analysis code. This also means if a user wants to bring their own dataset, none of the analysis functions depend on this specific DataFrame schema – they just require the appropriate inputs. The simulation code can also be extended to generate time-series data (e.g. outcomes over multiple days for novelty analysis). We might indeed extend `subscription_experiment` to output daily metrics for, say, 30 days for each user, with an initial novelty boost that decays. That could be an upgrade to simulate novelty effect explicitly for a lab exercise.

- `utils.py`**:** (Optional) We might not need a separate utils if the above are well-scoped. But typically, any helper that doesn't neatly fit elsewhere goes here. For example, computing a confidence interval given mean, SE, and alpha might be a tiny helper used in multiple places. Or a generic function to calculate Cohen's h/d (though we put those in power). We can create this if needed during implementation, but if it stays empty we can omit it.

Public API Design: With the modules in place, the **public API** (i.e. what functions a user of this package would call) will likely be a subset of the functions described above. We want to expose functions that are most generally useful. For instance:

- From `power.py`: `required_samples_binary` and `required_samples_continuous` would be key.
- From `srm.py`: `check_srm` (returning a p-value or boolean) is a staple check to run immediately after data collection. Possibly `baseline_balance_test` for any pre-ex metrics.

- From `frequentist.py`: `z_test_proportions`, `t_test` (Welch by default), and `mann_whitney_u` are core. We might unify interface by making them return a result object or a dict (with fields like `estimate`, `pvalue`, `ci`) for consistency.
- From `bayesian.py`: maybe `bayes_conversion_analysis(x_A, n_A, x_B, n_B)` that returns `prob_B_beats_A`, `posterior_diff_dist` (or its mean and HDI). Keep it simple for now (Beta-Binomial only).
- From `variance_reduction.py`: `cuped_adjust` (and perhaps directly a `cuped_t_test` that applies CUPED then does a t-test on adjusted outcomes), and `cupac_adjust` (which might internally train a model – we can allow passing in the model or model parameters).
- From `sequential.py`: perhaps expose `obrien_fleming_boundary(alpha, information_fraction)` and a helper `adjusted_p_value(original_p, current_look, total_looks)` if we implement alpha-spending in p-value form.
- From `multiple_testing.py`: `apply_bonferroni(p_values)` returning adjusted p-values or critical value, and `apply_bh(p_values)` similarly. Possibly a combined function `correct_pvalues(p_values, method="bonferroni")` for general use.
- From `ratio_metrics.py`: `analyze_ratio_diff(num_A, den_A, num_B, den_B)` that returns diff, CI, p-value for the ratio metric difference. The user just plugs in their metric numerator and denominator arrays for each group.
- From `noncompliance.py`: `itt_effect(y_A, y_B)`, `per_protocol_effect(y_A_exposed, y_B_exposed)` and `compute_cace(effect_itt, take_rate_B, take_rate_A)` or directly `estimate_cace(y_A, treat_A, y_B, treat_B)` that computes it from raw data (e.g. using 2SLS or formula).
- From `hte.py`: `x_learner_cate(X, y, treat)` returning individual CATEs and summary (ATE, etc.), and perhaps `segment_analysis(data, segment_col, metric_col, treat_col)` that returns a table of effect sizes per segment with p-values.
- From `diagnostics.py`: `check_guardrail(metric_name, diff, ci_low, threshold)` or something that evaluates if a guardrail metric's result is acceptable. Also `plot_metric_trend(metric_daily_values_A, metric_daily_values_B)` (if we include plotting capabilities) to visualize novelty – or a simpler `test_for_trend(effect_over_time)` which could do a linear regression on the effect vs. time to see if slope is significant. For novelty detection, a significant negative slope might indicate a novelty decay [27].
- From `reporting.py`: Possibly `make_decision(primary_result, guardrail_results)` that returns a decision recommendation (this could implement a more advanced logic than the simple matrix, e.g. "Ship if primary is positive & significant AND no guardrail significantly negative; if primary not significant but positive and guardrails fine, consider extending or need more data; etc."). Another could be `summarize_results(results_dict)` that prints out a nicely formatted summary for multiple metrics.

All these functions will have clear docstrings so that using the library in a notebook is straightforward (the user can call `help(ab_testing.power.required_samples_binary)` to see usage). By designing the API around functions (with possibly some lightweight classes for grouping results), we ensure someone could use this library in their own experimental analysis pipeline.

Notebook vs. Library Code: We will maintain both a **curriculum notebook** and a **script/CLI** entry point. The heavy lifting (computations) will reside in the library modules. The notebook will become a guided tutorial that *calls* these functions in sequence, rather than having raw code inline. For example, the notebook's section on "1.2 Z-test for Proportions" will call `frequentist.z_test_proportions(x_A,` `n_A, x_B, n_B)` and then pretty-print the result and interpretation. The narrative (explanation text) stays in the notebook (or script comments) for learning purposes, but the actual numbers come from the library. This ensures consistency (e.g. if we update how a CI is calculated, the notebook automatically uses the new method). The CLI script (perhaps `src/ab_testing/__main__.py` so that one can run `python -m ab_testing`) can similarly orchestrate a full run-through: generate data, run all analyses, and output to console. It might essentially reproduce what `ab_testing_framework.py` did, but in a loop calling each module. We'll likely keep the printouts and format similar, just refactored. This way, users have two interfaces: an interactive notebook for step-by-step learning, and a script that dumps the full analysis for a quick review or demonstration. Both will rely on the modular code under the hood.

With this modular setup and clear API, the repository transforms from a one-off script into a **mini experimentation library**. This not only looks professional but also enables easier extension (e.g. adding a new method like a false discovery rate control or a new type of test is as simple as adding a function in the right module, without touching a massive script).

Up-to-Date Experimentation Curriculum Map

We will reorganize and augment the curriculum to ensure it includes the latest best practices and a logical progression from fundamentals to advanced topics. Below is a map of the concepts, including *new additions* that will differentiate your expertise. For each concept, we detail (1) what it is, (2) when to use it, (3) common failure modes or pitfalls, and (4) a practical exercise idea for the repo. The overall flow remains in "levels", but we may reorder some topics for better coherence (e.g. discussing SRM right after assignment, before outcome analysis).

Level 1: Fundamentals

These are the must-know basics for any data scientist in experimentation. We ensure these are solid before moving on.

- **Sample Size & Power Analysis** – *What:* Calculating how many users you need in each group to detect a Minimum Detectable Effect (MDE) at a given power (usually 80%) [28]. Covers formulas for binary metrics (using conversion rate baseline and Cohen's h) and continuous metrics (using standard deviation and Cohen's d). *When to Use:* At experiment design time, before launching, to ensure the test is feasible (will you get enough sample in reasonable time?) and to communicate the duration to stakeholders. *Pitfalls:* Using the wrong inputs (e.g. guessing an MDE that's too optimistic or not business-relevant), not accounting for baseline variability, or confusing one-tailed vs. two-tailed in calculations. Also, forgetting that these formulas assume i.i.d. and no huge skew – if those assumptions break, power calcs can mislead. *Exercise:* We'll have an exercise where the student is given a baseline conversion and an expected lift (say 5%), and they must use the `power.required_samples_binary` function to compute needed sample size, then discuss how the duration would change if the lift were smaller. Another exercise: vary the power from 80% to 90% and see how sample requirements grow (to internalize the cost of higher power).

- **Randomization Checks (SRM and Balance)** – *What:* Tests to verify the experiment randomization was implemented correctly. SRM (Sample Ratio Mismatch) uses a chi-square test to see if the split of users is far from the intended 50/50 [14]. We will also emphasize checking that pre-experiment observable characteristics are balanced (no significant differences in means of pre-engagement, etc., beyond what chance allows). *When to Use:* Immediately after data collection, *before* analyzing outcomes. If SRM or imbalance is detected, it's a red flag that something went wrong (no point in analyzing lift, as the groups aren't comparable). *Pitfalls:* A common mistake is skipping this step and trusting the data blindly – leading to possibly basing decisions on a flawed experiment. If SRM is detected, pitfalls include jumping to analysis anyway (you risk analyzing garbage) or misidentifying the cause. We'll list common causes: bots or fraud affecting one group, a bug that assigned users unevenly, users not entering the experiment funnel uniformly, etc [8] . *Exercise:* We could simulate an SRM scenario: e.g. drop 10% of data from group B to mimic a tracking bug, and have the student run `srm.check_srm` to catch it. Another exercise might provide two small datasets, one with a true random split and one with an SRM, and ask the student to identify which is healthy.

- **Hypothesis Testing – Proportions (Z-test)** – *What:* The classical two-sample z-test for difference in proportions (e.g. conversion rates). We explain the null/alt hypotheses and why we use a **pooled** standard error under H_0 [18] , versus using unpooled for confidence interval. *When to Use:* For binary metrics (conversion, click-through, retention rates) when sample size is large enough for normal approximation. *Pitfalls:* Interpreting p-values incorrectly (the repo already warns that p-value is not "probability the treatment works"). Another pitfall: not checking assumptions (if sample is small or proportions extreme, the z-test may be less accurate – could consider Fisher's exact test in edge cases). Also, using a two-tailed test when you had a one-tailed hypothesis (or vice-versa) – always clarify the hypothesis direction in interviews. *Exercise:* Given conversion data for A and B, use `frequentist.z_test_proportions` to compute p-value and CI. Then have the student interpret: e.g. "p=0.04" – what does that mean in plain English? (Answer: if no real effect, there's a 4% chance of seeing such a difference [29]). We can also ask: if the CI for difference is [0.1%, 2.0%], what does that imply for business (e.g. "we are 95% confident the lift is between 0.1 and 2.0 percentage points")? This tests interpretation.

- **Hypothesis Testing – Means (Welch's t-test)** – *What:* Comparing two means (e.g. revenue per user, session length) using Welch's t-test which does not assume equal variance. We'll reinforce why Welch's is preferred in almost all practical cases (unequal variances are common, and Welch's is more robust) [18] . *When to Use:* Continuous metrics that are reasonably symmetric or for which mean is the metric of interest (e.g. average order value among purchasers). Use Welch's any time you'd consider a t-test, unless you have a strong reason variances are equal (rarely known in advance). *Pitfalls:* Non-normal or heavily skewed data – if the data are very skewed (like revenue with many zeroes and some huge outliers), the t-test might be formally significant but not very informative (hence non-parametric or bootstrap might be better – which we cover later). Another pitfall: forgetting that t-test assumes independence – if your data has clustering (e.g. multiple observations per user or some seasonality), you must handle that separately. *Exercise:* Using the revenue data from converters in the simulation, students will run `frequentist.t_test(revenue_A, revenue_B)` and get output. Then they'll compare this parametric CI to a bootstrap CI (from later section) to see differences. A question could be "Why might the t-test CI differ from the bootstrap CI for revenue? (Hint: revenue distribution shape)." Expected answer: because revenue is skewed, the t-test's normal theory CI might be less accurate, whereas bootstrap (non-parametric) captures the skew. This exercise ties fundamental and intermediate topics together.

- **Non-Parametric Test – Mann-Whitney U** – *What:* A test for whether one distribution tends to have larger values than the other, without assuming normality. We include it at Fundamentals or Intermediate level (currently in Level 2, but we might keep it in Intermediate). It's useful as a rankbased test for skewed data (e.g. revenue including zeros). *When to Use:* When the metric is continuous but violates assumptions of t-test (significantly non-normal, outliers, ordinal data, etc.), and especially when interested in median differences. Also if sample size is moderate and you want a test that is robust to outliers. *Pitfalls:* Interpreting the Mann-Whitney as a test of means – it's not exactly that (it tests P(X>Y) > 0.5 essentially). In an interview, one might ask "what if data isn't normal?" – knowing Mann-Whitney (or the bootstrap) as alternatives is key. Another caution is that Mann-Whitney can be less powerful if distributions differ mostly in variance not location. *Exercise:* The student will apply `frequentist.mann_whitney_u` on the revenue data and report the rank-biserial correlation (which we compute as an effect size). They should interpret it: e.g. "r = 0.12 suggests a small positive effect" and contrast that with Cohen's d from the t-test. This shows them how effect size interpretation can differ by metric and test.

- **Confidence Intervals & Effect Sizes** – *What:* Emphasizing estimation over just p-values. For each test above, we show how to derive 95% CIs for the difference and calculate effect sizes (Cohen's h for proportions [30], Cohen's d for means). *When to Use:* Always report CIs in addition to p-values – they convey the magnitude and uncertainty of the effect. Effect sizes help standardize impact (small, medium, large) in a way that's comparable across contexts. *Pitfalls:* Ignoring CIs (leading to overconfidence in a point estimate), or misinterpreting them (e.g. "we have 95% chance the true lift is in this interval" – which is a subtle misinterpretation; technically the CI either contains the true effect or not, but it's a long-run frequency statement). We'll still encourage the intuitive interpretation though. Another pitfall: relying on Cohen's cutoffs blindly without context (e.g. "small effect" might still be hugely valuable in business if baseline is low). *Exercise:* For a given result (say conversion lift), have the student interpret the CI: "If the 95% CI of the lift is [1%, 3%], what does that mean for our business decision?" (Expected: we're pretty sure the true lift is positive and at least 1%, so it's likely worthwhile; also the CI not crossing 0 aligns with a significant result). We can also give an example where p is >0.05 but CI is say [-0.5%, +0.5%] – not significant, but also effect is very small in either direction, which might suggest the feature truly does nothing important. This underscores that not significant *and* trivially small effect -> likely no impact, whereas not significant but wide CI -> we just lack data (could be meaningful effect that we missed). Understanding this nuance is differentiating.

- **Sanity: A/A Test or AA Simulation (optional)** – We might include a brief mention or exercise where an "A vs A" comparison is done to verify the methodology doesn't produce false positives beyond α. *What:* Splitting data into two pseudo-groups with no real difference (both using control variant data, for instance) and running tests should yield non-significance ~95% of the time at α=0.05. *When:* Usually done when validating an experimentation system or metric tracking (before running real experiments). *Pitfalls:* If an AA test shows significance, that indicates issues (either instrumentation or hidden bias or inflated false positive from multiple looks, etc.). *Exercise:* Take the control group data, randomly label half as "Pseudo-A" and half as "Pseudo-B", run a conversion rate test – students should see p ~ N(0,1) with about 5% chance <0.05. If they do this multiple times (we can have them loop 20 times to see how often a false positive arises), it reinforces the concept of false positive rate. This is more of a conceptual exercise to build intuition about randomness and p-value meaning.

(By the end of Level 1, the learner has designed the test, ensured it's set up right, and performed basic analysis of whether there is an effect, with correct interpretation. Next, we add sophistication to our analyses.)

Level 2: Intermediate Topics

These are expected knowledge for a data scientist who goes beyond the basics – they address common practical scenarios in experimentation and add rigor to analysis.

- **Multiple Testing Corrections** – *What:* Procedures like Bonferroni and Benjamini-Hochberg (FDR) to control false positives when examining multiple metrics or multiple variations. The repo currently demonstrates how examining 5 metrics inflates the chance of a false alarm to ~23% [30], and how Bonferroni (very strict) and BH (less strict) handle it. *When to Use:* Whenever you plan to make decisions with more than one hypothesis test in play – e.g. you have secondary metrics or you run several experiments in parallel on related things. In an interview, if you mention checking many metrics, they might ask "how do you avoid false discoveries?" – knowing these methods is key. *Pitfalls:* Bonferroni is easy but can be overly conservative (increasing Type II errors – missing real effects because the threshold is too high) [31]. BH controls expected false discovery rate and is more powerful, but one must be careful if metrics are not independent (it's still valid in expectation, but correlated metrics might make interpretation tricky). Another point: **guardrail metrics** – as per Spotify's research, if you use non-inferiority tests for guardrails, you might *not* adjust alpha for them [9]. We'll discuss that adjusting for guardrails can reduce power unnecessarily, because you usually only act if guardrails significantly worsen. They recommend focusing on controlling false negatives for guardrails (ensuring you have enough power to detect a deterioration) [9]. This is an advanced nuance we'll include in discussion. *Exercise:* We will present a scenario with, say, 4 metrics: conversion (primary), time on site, NPS, and retention (secondary/guardrails). The student will calculate how many of those show p<0.05 with no correction, then apply Bonferroni and BH using `multiple_testing.apply_bonferroni` and `apply_bh`. A question: "If conversion is significant raw p=0.04, but with Bonferroni it is not (0.16), what would you do?" Expected: Recognize that if conversion was the declared primary metric *a priori*, many teams would not apply Bonferroni to it – they'd control error at the experiment level by structuring hypotheses (this touches on the decision rule concept). But if all metrics are equally considered, then formally it's not significant under Bonferroni. This exercise highlights the importance of pre-specifying primary metrics.

- **Bayesian A/B Testing (Beta-Binomial)** – *What:* Framing the conversion rate comparison in a Bayesian way to get `P(B > A)` and credible intervals for the lift. We'll use Beta priors (likely uniform Beta(1,1) as non-informative) to update posterior for each variant's conversion rate [27]. Then compute the probability treatment is better and the expected loss of choosing each variant. *When to Use:* Bayesian analysis is useful when you want a more intuitive interpretation ("There's a 95% probability the new feature increases conversion"), or when continuous monitoring is needed (Bayesian posteriors are not impacted by optional stopping in the same way, given a proper framework). Many modern experiment platforms still rely on NHST, but being able to discuss Bayesian is a plus – especially for smaller sample scenarios or for providing a richer picture of uncertainty. *Pitfalls:* One must be careful not to misuse Bayesian results either – e.g. a high `P(B>A)` can be misleading if the credible interval is wide (it might just be a high probability of a tiny effect). Also, choice of priors matters: we'll demonstrate using a flat prior; but if you had informative priors or used Bayes factors, it could get complex. Another pitfall: **mixing Bayesian and frequentist** – e.g. you shouldn't apply frequentist sequential stopping rules to Bayesian posteriors; instead use decision theoretic thresholds. We'll mention that Bayesian methods, while more robust to peeking, still require decision criteria (e.g. stop when P(B>A)>95% *and* loss is below some threshold). *Exercise:* We'll have students use `bayesian.beta_binomial_posteriors(x_A, n_A, x_B, n_B)` to get posterior distributions, then have them simulate draws (or use our function) to estimate P(B>A) and the 95% credible interval for the lift. They will compare this to the p-value from the z-test. For example, if n is very large and difference is significant, P(B>A) will be ~ ~99% and the credible interval will exclude 0 – results agree. If n is small and p-value was ~0.1 (not significant), they might find P(B>A) ~ 80% with a very wide credible interval. This exercise shows how Bayesian can still "lean" one way (80% chance B is better) even when p>0.05, but with wide uncertainty. We'll ask: "Would you ship with P(B>A)=80% and why or why not?" (Expect discussion about risk tolerance; 80% might not be high enough confidence, credible interval likely crosses zero, etc. – a nice way to talk about practical decision-making beyond the dichotomy of p<0.05).

- **Bootstrap Confidence Intervals** – *What:* Using bootstrapping (resampling) to derive a confidence interval (and even a p-value if desired) without normal assumptions. Especially useful for metrics like revenue that are skewed or have outliers. We'll generate many resamples of A and B data and compute the difference in means for each, then take percentiles for the CI [28]. *When to Use:* When you suspect the sampling distribution of your metric difference is not well-approximated by a normal or when you don't have a formula for the metric's variance (like median or percentile metrics). Also, in interviews, mentioning bootstrap is great when asked "what if data isn't parametric?" *Pitfalls:* Bootstrapping can be computationally heavy (resampling 10,000+ times). It also assumes the sample is representative of the population (so it works better with larger samples). A subtle pitfall: if data has structure (clusters, time series), you need a more advanced bootstrap (block bootstrap, etc.) – our context is simple random sample though. Also, people often misinterpret bootstrap CIs similarly to standard ones – the interpretation is essentially the same (just constructed differently). *Exercise:* We'll have the student implement or use our `frequentist.bootstrap_ci` (if we provide one) on the revenue metric and compare with the t-test CI. They might see the bootstrap CI is asymmetric (if we look at median or so). For example, if revenue lift is heavily influenced by a few whales, bootstrap might show a very wide upper bound. We ask: "Why is the bootstrap CI for revenue so skewed?" (Expected: because the distribution is skewed, a few high values in some resamples bump up the mean difference). We could also ask them to bootstrap the median difference (which no closed-form exists for easily) to illustrate bootstrap's flexibility.

- **Non-Parametric Effect Size – Rank-Biserial (for Mann-Whitney)** – This is a minor concept but we'll mention it for completeness. *What:* Rank-biserial correlation is an interpretation of the U test result as a measure of how often treatment values exceed control values. *When to Use:* Reporting effect size for Mann-Whitney tests since Cohen's d doesn't apply. *Pitfalls:* Many don't understand this metric, so if communicating to non-stats folks, you might stick to median differences or so. It's fine to include as enrichment. *Exercise:* Already covered above within Mann-Whitney exercise – compute and interpret.

- **Intent-to-Treat (ITT) vs. Per-Protocol (PP)** – *What:* ITT means analyze everyone as originally assigned, regardless of whether they fully received the treatment. PP (or "treatment-on-treated") means analyze only those who actually got the treatment (in our simulation, the subset with `triggered=1`). We demonstrate that PP will show a larger effect size if some in treatment never saw the feature (dilution effect). *When to Use:* Always calculate ITT as the primary analysis (because it preserves randomization and is unbiased for rollout decision). Use PP as a secondary analysis to diagnose *why* an effect might be smaller than expected – it tells you the potential effect if everyone got the treatment (useful for estimating upside if implementation issues are fixed). In practice, a big gap between ITT and PP suggests a rollout problem (e.g. many didn't see the feature or didn't engage with it). *Pitfalls:* Doing only PP analysis – that can introduce bias because those who comply may differ systematically from those who don't (breaking randomization). Also, interpreting PP causally is tricky unless you account for selection (this is where IV comes in). *New Addition:* We will introduce the concept of **Encouragement Design and Instrumental Variables** here. We'll explain that our experiment is an "encouragement" – users were encouraged to see new onboarding (if in B) but some didn't. The proper causal effect on compliers (those who comply with encouragement) can be calculated by **Instrumental Variable** formula: LATE = ITT / take-up rate [10]. We'll mention the assumptions (no defiers, exclusion – i.e., being assigned B only affects outcome through actually using the new onboarding) [32] [33]. We won't deep-dive into IV math in code beyond possibly using our `noncompliance.compute_cace`. But conceptually, this is **very differentiating**: few candidates bring up instrumental variables in an A/B test context, so it will impress if you can mention "we calculated the CACE using assignment as an instrument to account for the 10% of users who never saw the treatment." We'll cite that this is exactly what Spotify describes in their engineering blog [11] [12]. *Exercise:* Students will compute ITT and PP conversion lift using our functions. Then, using the observed trigger rates (100% in A, 90% in B), they will compute the implied CACE = ITT / 0.9. We'll ask: "Does the PP lift match the CACE?" (It should closely). Then a thought question: "If you were product manager and saw that ITT lift is 2% but PP lift is 2.5%, what does that tell you?" Expected: 20-25% of the potential impact is lost due to non-compliance; we might investigate why 10% of users didn't see the new onboarding (technical issues? user skipped?). It indicates an opportunity: if we ensure everyone sees it, we could maybe get a 2.5% lift. But we'd also consider that 0.5% might be due to those who didn't see it presumably having lower conversion anyway (maybe they dropped off early). This segues into discussions on improving experiment delivery.

By Level 2's end, you will have covered multiple comparisons, Bayesian thinking, non-parametric methods, variance reduction (coming next), and advanced analysis splits. These are standard toolkit items in many DS roles.

Level 3: Advanced Techniques

These topics set you apart as someone who understands the cutting-edge of experimentation and can handle complex scenarios. They often address problems of reducing variance, handling peeking, and extracting more insights (like who does it work for).

- **Variance Reduction – CUPED** – *What:* Controlled Using Pre-Experiment Data. We subtract out variation explained by a pre-experiment covariate (e.g. prior month's revenue or engagement) from the outcome. This can drastically reduce variance if the covariate correlates with the outcome [15] [34]. *When to Use:* When you have a reliable metric from before the experiment that is correlated with your outcome, and especially if your metric of interest has high variance. It's commonly used in metrics like revenue, engagement time, etc., to improve sensitivity. Best used when that covariate is not affected by the treatment (since it's from before, that's ensured). *Pitfalls:* If the covariate is not truly independent of treatment assignment (e.g. if there was some unintended bias or if the experiment is long and the covariate changes due to external factors in a correlated way), CUPED could introduce bias. Also, implementing CUPED means you need that historical data readily available for your analysis pipeline. In interviews, a common mistake is to say "CUPED always helps" – if the covariate is weakly correlated, you gain little or nothing (could even lose a degree of freedom). We'll quantify variance reduction ≈ ρ^2 (correlation squared) [35]. *Exercise:* We'll have them apply `variance_reduction.cuped_adjust` on revenue. They will compute the standard error of the difference *with* and *without* CUPED and see the reduction. For example, if pre-period revenue has correlation ~0.5 with experiment revenue, we might see ~25% variance reduction (since ρ^2 ~0.25) – our simulation actually prints this out. We can ask: “CUPED adjusted CI vs raw CI – which is narrower and by what percent?” (Expect an answer like “CUPED CI is ~X% narrower, meaning we'd need ~X% fewer samples for same power”). The exercise reinforces how pre-data translates to sample size savings.

- **Variance Reduction – CUPAC (ML-based)** – *What:* An extension of CUPED where instead of a single covariate, you use a Machine Learning model to predict the outcome using many features (pre-experiment behavior, demographics, etc.), then use that prediction as a composite covariate [16] [36]. It captures nonlinear relationships and interactions, potentially reducing variance even more (DoorDash reported 15-20% extra gain over traditional controls and ~40% reduction in test duration overall) [17]. *When to Use:* In high-variance metrics where you have rich user data. For example, in fintech or e-commerce, user spend can vary wildly; using ML to predict spend (or propensity to convert) from historical data can control for that variation. It's most useful when a single covariate isn't enough or in scenarios like recommendation systems where prior behavior predicts future outcomes. *Pitfalls:* Overfitting in the covariate model – we must use techniques like cross-fitting (out-of-sample predictions) to avoid biasing the treatment effect. Also, more computational overhead and complexity (maintaining an ML model for each experiment). In practice, one must ensure the model's features are **not influenced by treatment** (they should all be pre-treatment or unrelated). Another subtle pitfall: model leakage – if your model accidentally includes a feature that is affected by treatment (which can happen if your training data spans into experiment period by mistake), it would invalidate the adjustment. *Exercise:* We'll let students use `variance_reduction.cupac_adjust` on the same revenue data, which internally trains a GradientBoostingRegressor on pre_engagement, pre_revenue, etc. They will compare the variance reduction from CUPAC vs CUPED. Likely, CUPAC will show a further drop in variance (our simulation might achieve, say, 30-35% reduction). We ask: “Why did CUPAC reduce more variance than CUPED?” (Expected: because it leveraged multiple pre metrics and nonlinear relations, capturing more of the outcome variance [37]). Also: “What's the trade-off of CUPAC?” (Answer: complexity – need to train model, risk of overfitting, harder to explain to stakeholders than a simple adjustment).

- **Sequential Testing (Early Stopping with Error Control)** – *What:* Methods to allow peeking at results or stopping an experiment early while still maintaining the overall Type I error $\leq \alpha$. We'll explain the concept of an **α-spending** approach like O'Brien-Fleming: very stringent significance thresholds for early looks, becoming more lenient later [3]. For example, at 3 equal interim looks for $\alpha=0.05$, the boundary might be $p<0.003$ at first look, 0.014 at second, 0.046 at final, etc. *When to Use:* When running long experiments where you want the option to stop early for success or futility. Many modern platform tools (e.g. Statsig, ABsmartly) offer sequential testing dashboards. If asked “what if executives want to peek at results weekly?”, you can say “we can use group sequential tests to account for that, e.g. using O'Brien-Fleming boundaries, so that the false positive rate is controlled” [29]. *Pitfalls:* Using naive $p<0.05$ at each peek – which inflates false positives dramatically [30]. Another issue: repeated looks increase the chance of **stopping at a random high** (volatility). Also, sequential tests usually assume a maximum number of looks and often equal spacing – deviating from plan can break guarantees. In practice, one must also consider the **cost of stopping early** in terms of power – boundaries like O'Brien-Fleming are conservative early, meaning you need a very strong early signal to stop. If you stop for futility, you might miss a late trend reversal (so called “slow burn” effects). *Exercise:* We will simulate an experiment outcome trajectory: say we pretend we can check results every week for 4 weeks. We’ll use `sequential.obrien_fleming_boundary` to get the Z or p thresholds for each look. Students will see that at look1 the threshold might correspond to p ~0.005. We then ask: “At week 2, the p-value is 0.02. Should we stop the test for significance?” Using O’Brien-Fleming, probably **No** (because maybe threshold is 0.015 at mid-point). This teaches them that even p=0.02 might not be enough if you promised to peek multiple times. It underscores why just reaching 0.05 early doesn’t mean you’ve won. We might also do a quick demonstration: generate a cumulative result where at some early look it would have been significant by naive p, but not by the corrected boundary – and indeed later it regresses. This concept can be tricky, but even awareness of it is a plus.

- **Heterogeneous Treatment Effects (HTE) – X-Learner & Segments** – *What:* Techniques to estimate and detect if the treatment effect varies across users. We’ll introduce HTE via two angles: (a) **Pre-defined segments**: e.g. does the effect differ on Organic vs Paid users, or iOS vs Android? We compute those differences with interaction terms or separate tests. (b) **Meta-learners (X-Learner)**: using machine learning to estimate individual CATE. The X-Learner we implemented uses two regression models and then a second stage [24] [38]. It provides an estimate of treatment effect for each user which we can aggregate or analyze distribution of. *When to Use:* After finding an overall effect (or even if not, to see if maybe some subset had positive effect that canceled out). Particularly used in personalization: if some groups benefit and others are hurt by a change, you might want to roll it out targeted to those who benefit. Also useful for post-experiment analysis to generate hypotheses (e.g. “it seems power-users responded negatively, while new users loved it”). *Pitfalls:* **Multiple comparisons** – the more segments you check, the more likely you’ll find a “difference” by chance (so apply corrections or keep it exploratory). For ML approaches, pitfalls include overfitting (especially if effect heterogeneity is weak, the model might just fit noise), and interpreting CATE casually – one must remember causal interpretation requires the features used adequately capture why treatment effect would differ (no hidden interactions). Another pitfall: sample size per segment – if some segments are small, their “effect” could be noise. *Exercise:* We’ll do two things: (1) Have students use `hte.segment_effects(df, 'channel', 'converted', 'variant')` to see conversion lift by acquisition channel. They’ll get results maybe like: Organic +5pp, Paid +1pp, Social +0pp, Referral +4pp (for example). We ask: “Which segment saw the highest lift and what might explain that?” (E.g. maybe Organic users had more room to improve or the feature was tailored to them). (2) Have them run `hte.x_learner_cate(X, y, treat)` with X = [pre_engagement, pre_sessions] as features. They’ll then inspect the distribution of CATE (maybe plot deciles). The exercise could ask: “According to the X-learner, what is the ATE for the population and how does the effect vary?” If our simulation built in heterogeneity (it did: e.g. Referral channel gets 30% more effect), the X-learner should pick something up. We might see, for instance, that users with high pre_engagement had bigger lift than those with low (since maybe the personalization worked better for engaged users). We’ll have them verify if the segments they found manually (e.g. by channel) align with what the model indicates. This showcases that they can uncover hidden patterns. *Differentiator:* Being able to discuss HTE modeling is a big plus – few junior DS know about meta-learners. Citing that “we can use techniques like X-Learner or causal forests to identify segments with different responses” will stand out. We can mention that Statsig recently talked about automated differential impact analysis [39] – so this is cutting-edge but increasingly expected in mature experimentation teams.

- **Quantile Treatment Effects** – *What:* Looking at how the treatment effect might differ at different points in the outcome distribution, not just the mean. For example, did the feature increase revenue for the top 10% spenders but not for others? We can compare quantiles (percentiles) of the outcome for treatment vs control. *When to Use:* When distributional impact matters – e.g. a feature might only benefit heavy users, or it might reduce variance (like it helps the low performers catch up). It's also useful for metrics like load time, where you care about tail improvements (p90 latency). *Pitfalls:* Harder to get statistical significance for quantile differences (there are quantile regression methods, but we can keep it simple by just observing differences). Also, one must be careful: if you see differences at, say, the 90th percentile, that could be because of outliers – tying it to a story is needed. *Exercise:* We already compute quantiles of revenue in the script for each group. We'll present a table of, say, 10th, 50th, 90th percentile of revenue for A vs B. Students will observe something like "At the 90th percentile, treatment users have $X more revenue than control, but at the median there's no difference" (implying the effect came from big spenders). We'll ask: "What does this tell you about the treatment's impact?" (Expected: it mainly increased revenue among the top spenders; maybe the feature was appealing to power users but didn't convert more new customers). We could also mention formal quantile regression if curious, but likely not implement. This concept is not mainstream for all interviews, but it's a good thinking framework: always ask "did the treatment help just the top users or everyone?" especially for skewed metrics.

- **Delta Method for Ratio Metrics** – *What:* Using the delta method to get the standard error for metrics that are ratios (like conversion = #conversions/#users, or ARPU = revenue/users, or CTR = clicks/impressions) [40]. We already have this in the code. It ensures we propagate uncertainty correctly. *When to Use:* Any time you have a derived metric that's a ratio of two random variables. A common scenario is "Revenue per user" or "Sessions per user" – you can't just do a t-test on per-user values if many users have zero; and you can't just compare two ratios as if they were means of something independent. The delta method leverages the counts of numerator and denominator to compute an approximate variance. *Pitfalls:* If denominator is very small or distribution is highly skewed, delta method (first-order Taylor expansion) might not be super accurate. Also it assumes independence of numerator & denominator (which for metrics like CTR might not strictly hold – e.g. users with more impressions might also have more clicks, introducing covariance; but the delta formula accounts for covariance term if you supply it [41]). Another pitfall is not recognizing when to use it: e.g. some might just take means of per-user CTR, which is okay if each user's weight is equal, but if impression counts differ a lot, the aggregate CTR variance needs careful handling. *Exercise:* We'll have a small exercise: Given overall revenue in A and B and user counts (or using our data, summing revenue and count), compute the difference in revenue per user and its CI using `ratio_metrics.analyze_ratio_diff`. Then compare that to if you naively treated revenue per user as an ordinary metric by taking mean of per-user revenue (which in our case is the same calculation because each user contributes one data point – a better example might be "click-through rate" where each user has different impressions). We can simulate a CTR scenario: say A had 100 clicks out of 1000 impressions, B had 120 clicks out of 1000 impressions. The delta method would consider the variance due to impressions. The exercise: "Use the delta method function to test if the difference in CTR (10% vs 12%) is significant." This reinforces how to properly compare rates. It's an advanced technique that many basic practitioners overlook – mentioning it will show statistical maturity.

- **Winsorization (Outlier Management)** – *What:* Capping extreme values (e.g. top 1% and bottom 1%) instead of removing them, to reduce the influence of outliers on mean and variance while not entirely discarding data. Our script did 1% winsorization on revenue. *When to Use:* When a metric has a heavy tail and a few extreme values are causing very large variance. Often used for revenue per user, session durations, etc., especially in reporting (so one huge outlier doesn't skew the average). It's also useful as a sensitivity analysis: "If we winsorize at 99th percentile, do results change much? If yes, maybe the effect was driven by a few outliers." *Pitfalls:* It can be seen as arbitrary or p-hacking if done post hoc; ideally the winsorization rule is set beforehand (e.g. we routinely cap at 99th percentile for metrics X, Y). Also, it biases the mean slightly (bringing it towards median), so one should interpret results carefully. But if done symmetrically and at extremes, bias is usually minor relative to variance reduction. Another caution: outliers might be real and important (e.g. one user spending \$10k might be a real whale – capping them could hide a segment effect). So use this mainly to improve statistical detection, but also examine outliers separately. *Exercise:* The student will apply `frequentist.t_test` on raw revenue vs. on winsorized revenue arrays and compare the p-values and CIs. They'll likely see the winsorized version has a smaller standard error (in our data, maybe a slightly tighter CI) – possibly turning a barely non-significant result into significant, or just reducing noise. Question: "After winsorizing, the mean difference changed from \$X to \$Y. What does that imply about the outliers' effect?" (Expected: if Y < X in absolute, the outliers were amplifying the measured effect; if Y > X, outliers were dampening it – usually outliers add noise, so often the effect becomes clearer after). We can also ask: "Should we always winsorize? When is it appropriate?" to prompt thinking on pre-registration and metric definitions.

At the end of Level 3, the user is exposed to methods that are used by top experimentation teams (variance reduction to squeeze more power, sequential testing to allow flexibility, and digging into for whom and in what way the treatment works). Mastering these will set them apart from someone who only knows basic A/B tests.

Level 4: Production & Beyond

This level ties everything into the real-world context of decision-making and scaling experimentation in a business.

- **Decision Framework & Guardrails** – *What:* A structured approach to decide **Ship, No Ship, or Need More Data** based on experiment results, taking into account the primary metric and guardrail metrics. The simple matrix (Significant? Positive?) from the repo is a start. We will enrich it by explicitly including guardrails: e.g. **Ship** if primary metric is significantly positive **and** no guardrail metrics significantly deteriorated; **Don't Ship** if primary metric is significantly negative (feature clearly hurts) or any guardrail shows significant harm beyond acceptable threshold; **Continue or Abandon** in grey cases (e.g. primary not significant but point estimate is positive – maybe continue if practical, or abandon if it's small and insignificant). Essentially, we align with the idea of **risk management in decisions** [42] [43]. *When to Use:* Every experiment review meeting! It's how you turn statistical findings into go/no-go decisions. *Pitfalls:* Not having a preset decision rule can lead to cherry-picking ("metrics moved in conflicting ways, but we'll focus on the positive one and ship" – that's risky and inconsistent). However, one must also be careful not to blindly follow a matrix if context dictates otherwise (e.g. a very slight significant negative on a secondary metric might be tolerable if the primary gain is huge – maybe you mitigate the downside separately). Another pitfall is ignoring **practical significance**: we should incorporate "is the lift big enough to matter?" into the decision (e.g. a 0.1% lift that's stat sig might not be worth the engineering effort to roll out, whereas a 5% lift not sig due to sample size might still be promising). *Exercise:* We'll pose a scenario: Primary metric conversion lift = +2% (p=0.03), guardrail metric retention = -1% (p=0.20, not significant). Ask, “What would you recommend?” According to a decision framework: the primary is a win, guardrail didn’t significantly drop (p>0.05), so likely **Ship**, but monitor retention in rollout (since point estimate is -1% even though not sig). Another scenario: primary +1% (p=0.15, not significant), guardrail 0 change – likely **Hold** and gather more data (not significant yet, but trend positive, maybe extend test). We will incorporate these reasoning exercises so the user practices making calls like they would in a real experiment debrief.

- **Business Impact Translation** – *What:* Converting the experiment metric lift into tangible business terms: annualized revenue impact, additional users converted, etc. The repo already does an annualized calculation (effect * userbase * LTV) which is excellent. We will keep that and perhaps add more nuance: e.g. calculating confidence bounds on the revenue impact (to communicate uncertainty). *When to Use:* In communicating results to executives or non-technical stakeholders. It answers “So what? How does this 2% conversion lift affect our bottom line?” *Pitfalls:* Over-extrapolation – e.g. assuming the lift will remain the same over a year, or generalizing from an experiment segment to all users without caution. Also, double-counting if multiple experiments impact the same users (in a portfolio sense). But those details aside, the main pitfall is *not* doing this – many DS present percent lifts and p-values, but fail to say “this will bring in roughly \$500K extra per month” which is what leaders care about. *Exercise:* The student will be asked to take a given lift (say +3% purchase rate) and given an annual traffic or user number and average revenue per user, compute the expected revenue increase per year. We’ll have them do this with our example: e.g. +0.5 percentage point conversion on 10 million users with \$150 LTV means ~50k extra conversions = \$7.5M/year [44] [45]. They should report a sentence as if to leadership. This will reinforce their ability to translate statistical results into business impact succinctly.

- **Experiment Lifecycle & Culture (qualitative)** – We might add a short discussion (in the README or notebook text) about the importance of **pre-registering hypotheses, experiment design alignment with decision rules, and post-test analysis**. Topics: how to pick a good primary metric (aligned to North Star), how to choose guardrails relevant to your product (as Mixpanel blog described: guardrails protect against unintended consequences in other areas [2]). Also, the idea of running **dry runs or shadow tests** (where you run the experiment with no change to see if any metrics move – a form of AA test for metrics integrity). Another advanced concept is **“Twyman’s Law”** – “Any figure that looks interesting or different is usually wrong” – i.e. be skeptical of extreme results until verified (often they’re instrumentation issues). Also emphasize a culture of learning: an experiment that doesn’t achieve a lift isn’t failure if you learn why or identify a segment where it did work. These are softer concepts but speaking to them shows maturity. We won’t necessarily have code for these, but we’ll update the curriculum narrative to mention them.

- **Optional: Always-On Experimentation & Bandit Strategies** – If space permits, we can briefly mention that not all optimization is done via fixed-horizon A/B tests. Some companies use **multi-armed bandits** to continuously allocate traffic to better variants (trading off exploration/exploitation). We won’t implement a bandit here (that’s quite complex and beyond classical A/B), but mentioning it along with pros/cons (e.g. bandits maximize reward but don’t give as clean an estimate of the effect as an A/B test, and they assume stationarity) could show you are aware of the broader landscape. This would be just a paragraph in the README or advanced section.

By restructuring and adding these concepts, the curriculum becomes very up-to-date. Notably, the additions of **interference (cluster experiments)**, **instrumental variables for noncompliance**, **guardrail metric decision rules**, and **novelty effect analysis** are directly drawn from recent industry discussions and papers (Meta on network effects [3], Spotify on IV [10] and multi-metrics [9], Statsig on novelty [46]). These will differentiate you. Many candidates might know CUPED or sequential testing, but fewer will bring up interference or IV unless they have real-world experience – showcasing these in your project (even if just conceptually with a citation and small demo) signals a higher level of understanding.

Each concept above will be incorporated into either the new modules or at least in the notebook narrative with a citation to sources like the Spotify or Meta engineering blogs for credibility. For example, when we mention cluster experiments, we'll cite Meta's blog showing how failing to isolate groups can shrink effects [3]. When discussing novelty, we'll quote Statsig's point that novelty effects are temporary and should be examined via time series [5]. This not only strengthens the content but also shows you've done your research (which is exactly what an interviewer or reader will notice – that you're citing Spotify's 2024 approach or a 2023 blog in your learning, which is impressive and shows curiosity).

Practice & Learning Roadmap

To solidify these concepts and demonstrate your skills, here's a proposed **practice plan**. It spans an initial 2-week burst to restructure and cover fundamentals, and a 6-week extended plan to master advanced topics and polish the project. Each week has specific deliverables to keep you on track:

First 2 Weeks (Core Refactoring & Fundamentals)

- **Week 1: Repository Refactor and Core Modules**
- *Goal:* Complete the modularization of the code and ensure all fundamental analyses still work as before.
- **Tasks:**
  - Set up the package structure (`ab_testing/` modules) and move code from the monolithic script into the appropriate modules (`power.py`, `frequentist.py`, `srm.py`, `simulation.py`, etc.). Make sure to include docstrings and correct citations in comments where appropriate.
  - Implement unit tests for the most critical functions: e.g. test that `power.required_samples_binary` returns known values for a given input (we can calculate a small scenario by hand or use statsmodels to verify), test that `srm.check_srm` correctly flags a known mismatch case, test that `frequentist.z_test_proportions` yields the same p-value as `statsmodels` or SciPy for a sample input, etc.
  - Re-run the original analysis (perhaps via the CLI script or notebook) to verify that results match the pre-refactor outputs (within rounding error). This means Level 1 outputs (sample size calcs, z-test, t-test, SRM) should be identical.
- **Deliverables:**
  - A reorganized repository committed to GitHub. The tests (at least for core functions) should pass locally.
  - An updated README or documentation section describing the new module structure and how to run the analysis.

    - You should also write a brief description of the refactoring in the commit or PR (this shows you understand and can communicate code structure changes).

- **Week 2: Fundamentals Deep Dive & Exercises**

- *Goal:* Reinforce understanding of Level 1 concepts and create practice content for them.
- **Tasks:**
    - Create a **"Fundamentals Lab" Jupyter notebook** (or a section in the main notebook) with prompts and placeholders for key calculations. For example, a question: "Compute the required sample size for a baseline conversion of 5% and MDE of 10% relative lift." and then the code cell expects the student to call the function and maybe interpret the result. Do this for power analysis (binary & continuous), running a z-test, and interpreting an output.
    - Add an exercise for SRM: perhaps intentionally modify a copy of the data to simulate SRM and ask the user to detect it. Also, include a question about what could cause SRM, referencing the list we have (bot traffic, etc.).
    - Ensure the notebook includes answers or at least guidelines after each exercise (could be hidden or in an appendix). The idea is for you to be able to practice and also show interviewers that you didn't just memorize outputs – you can actually calculate and reason.
    - Study each fundamental concept's theory a bit deeper: e.g. review "why use pooled proportion in z-test" so you can articulate it. This isn't a coding task, but a learning task – maybe write a short paragraph for yourself for each (power, z-test, t-test, SRM) that you can use as a speaking note. This will help internalize talk tracks.
- **Deliverables:**
    - The "Level 1 Fundamentals Lab" notebook with a set of Q&A or fill-in-the-blank style exercises completed. Push this to GitHub as well.
    - A short write-up (could be in README or a separate Markdown) summarizing the key points learned in Level 1, in your own words. This could even be a blog post or documentation page – something you can quickly review before interviews.

(By end of Week 2, you'll have a clean repo and solid command of the basics, with evidence of practice via notebooks.)

Weeks 3–6 (Advanced Topics, Project Completion, and Capstone)

- **Week 3: Intermediate Methods & New Additions**
- *Goal:* Implement and practice Level 2 topics, including new content like multiple testing guardrails and Bayesian analysis.
- **Tasks:**
    - Finalize functions in `multiple_testing.py` and create a small **Multiple Testing Lab**. This lab might show a scenario of multiple metrics and have the student apply Bonferroni and BH. Also, explicitly include a question about guardrail metrics: e.g. "If metric X is a guardrail, should we adjust its alpha? (Hint: guardrails use non-inferiority; Spotify says no FP adjustment [9], but ensure high power to detect downsides)." This will force you to articulate that concept.
    - Flesh out `bayesian.py` and test it on the data. Perhaps compare the Bayesian conversion result to the frequentist one in code. Create an exercise: given the posterior distributions from an experiment, interpret P(B > A) = 94% – should we launch? (This has no single answer,

    but you can provide reasoning: 94% is evidence in favor but maybe you want >95% or consider the cost of a wrong decision).
  - Integrate a section on **novelty effects**: simulate or assume the treatment effect decays over time. For practice, you could take the daily metrics (if you extend the simulation to output daily conversion rates for a couple of weeks). Plot or calculate the effect each day. If you don't have actual time series, you can hypothetically break the data (e.g. first 50k users as "Week 1" cohort, next 50k as "Week 2" – not perfect but illustrative if we impose a difference). Ask: "Does the lift increase, decrease, or stay constant over time?" If you see it decreasing, that's a novelty effect sign. Write down how you'd explain novelty: e.g. cite Statsig "response to new feature is temporarily inflated due to newness, might wear off" [5] .
  - Continue developing unit tests or checks for these new functionalities (e.g. test that Bonferroni function works on a known example, test that Bayesian posterior means match manual calculations for a small input).

- **Deliverables:**

  - Updated code for intermediate functions (multiple testing, Bayesian etc.) along with at least one small notebook or an expanded main notebook section demonstrating them on the simulated data.
  - Written notes or slides (maybe in the project wiki or as a PDF) summarizing intermediate topics in plain language – you can pretend this is for a study group or for a blog post series. This further cements your understanding and gives you materials to review later.

- **Week 4: Advanced Methods Implementation**

- *Goal:* Complete implementation and understanding of Level 3 advanced techniques (CUPED, CUPAC, sequential testing, heterogeneity, etc.).
- **Tasks:**
  - Program the `variance_reduction.cuped_adjust` and `cupac_adjust` functions and test them on the dataset. Ensure CUPAC uses out-of-fold predictions. Possibly simulate a simple check: if you use outcome itself as covariate (silly test), variance should drop to ~0 – to verify the adjustment logic. Write tests for CUPED (like if X and Y are perfectly correlated, output variance should reduce to 0 ideally).
  - Create an **Advanced Variance Reduction Lab**: Provide the pre and post adjustment variances and ask the student (yourself) to calculate the variance reduction %. Also ask conceptual questions: "If pre and post correlation ρ = 0.6, roughly how much variance reduction do we expect? (Answer ~36%)."
  - Implement `sequential.obrien_fleming_boundary` and test with a known table (some literature values, or verify that as looks →∞, boundary → ~1.96). Perhaps simulate a scenario in code where you peek and see how unadjusted vs adjusted decisions differ.
  - For HTE, finalize the `hte.x_learner_cate` function. This is complex; test it on a known simple scenario: e.g. construct data where the true treatment effect is +5 for one subgroup and 0 for another, see if the learner roughly recovers that. If using econml is permissible, you might double-check results against it for validation.
  - Add an **HTE Lab/Exercise**: maybe have the student identify which feature (pre_engagement or device or channel) seems to drive the largest difference in CATE. Also, provide a scenario: "Suppose the X-learner found a subgroup of 5% of users with a very high positive CATE, and some with negative CATE. How might you use this info?" (Expected: target the high positives

    for rollout, or investigate why negative for some – perhaps you wouldn't launch to them). This makes you think about business actions from HTE analysis.
  - Consider interference: While we may not fully simulate a cluster experiment, at least write a question in the lab or README: "What if users in control and treatment interact? What design could we use?" (Expected answer: cluster randomization – group users into clusters (e.g. networks) and treat entire clusters [4], at the cost of higher variance due to ICC [47] [48]). Just including this thought and answer (with citation) in your materials shows you're aware of the interference solution.

- **Deliverables:**

  - All advanced features coded and either demonstrated in the main notebook or separate labs. This likely includes sections in the notebook for CUPED/CUPAC comparison, sequential test example, and HTE results.
  - By this point, the main notebook might be quite large. Ensure it's well-organized with a TOC. Alternatively, break it into smaller notebooks per level to manage size (just note that in README for navigation).
  - A set of Q&A notes for advanced topics, similar to prior weeks. You might draft potential interview questions like "How would you improve the power of an experiment without increasing sample size?" and practice answering (CUPED/CUPAC, using stratification, etc., with references to what you implemented).

- **Week 5: Project Polishing and Documentation**

- *Goal:* Turn the repository into a polished portfolio piece with clear documentation, tests, and an attractive README. Also rehearse explaining each part.
- **Tasks:**
  - Write a comprehensive **README.md** that now reflects the modular structure and updated content. It should start with a high-level overview (what this project is, scenario, key techniques), then perhaps a table of contents of concepts. It should tell someone how to run the analysis (e.g. "`uv run jupyter lab` to open the notebooks, or `uv run python src/ab_testing` to see console output"). Also, include references to the sources you've drawn on, to show you incorporated industry knowledge (you can have a references section with links to the Spotify blog, Meta article, etc. – this adds credibility).
  - Ensure the README or a separate docs file includes the curriculum map from above, either in prose or a list, so anyone (like an interviewer) can see the breadth of what you learned.
  - Set up a lightweight **CI pipeline** (if using GitHub, maybe a GitHub Actions workflow for running tests on push). Use `pytest` to run your unit tests, and possibly `ruff` or `flake8` for linting to show code quality. This doesn't directly contribute to experimentation knowledge, but it showcases engineering rigor. Plus, it will catch any cross-platform issues or undeclared dependencies.
  - Go through the entire project as if you're presenting it. This means: open the notebook, run all cells, ensure plots (if any) render, ensure outputs aren't too verbose (maybe trim some lengthy outputs or move them to appendix). Practice narrating the story: "First, we did power analysis to ensure our test is sized correctly... Then we ran a hypothesis test and found significance... We checked multiple metrics and saw one potential issue with NPS (though not significant after correction)... We applied CUPED to see if we could have saved users... etc." Being able to tell this story smoothly is the goal by end of week 5.

    - Address any outstanding issues or TODOs in code. If some tests are failing or if certain advanced functions are too slow (bootstrapping 10k can be slow – maybe reduce to 5k in demo), fine-tune them now.

- **Deliverables:**

    - A polished README and possibly a docs folder with additional write-ups (could include the exercises solutions, etc.).
    - GitHub Actions (or similar) showing a passing build on your repo. This is visible on GitHub and signals that your project is not just a static analysis but an actively maintained piece of software.
    - Finalized notebooks with all outputs cleared (or kept, depending on preference) and ready for viewers. Clearing outputs before pushing can be good so others run it themselves; but having outputs is fine if they're static. Just be consistent.

- **Week 6: Capstone Project and Presentation Practice**

- *Goal:* Do a realistic end-to-end analysis (possibly on a fresh dataset or a variation) using the toolkit, and prepare a concise readout as if to stakeholders.
- **Tasks:**
    - **Capstone analysis:** Use your framework on a new scenario. This could be: modify the simulation to a slightly different experiment (e.g. simulate an email A/B test with a different metric), or take an open dataset (if any exists for experiments – though that's rare). Alternatively, you can take the same data but pretend only a subset of metrics matter to a different stakeholder, etc. The idea is to demonstrate using your library in practice quickly. For example, simulate a smaller experiment (n=5000) where B actually has no effect, run through the analysis steps and see how results look when nothing is significant – so you practice the narrative of "we found no significant differences; power was X so it's possible we missed a small effect, etc.".
    - Create a short **slide deck or report** of this capstone experiment as if you were presenting to product leadership. Focus on clarity: one slide on goal & design (include the sample size calc briefly), one slide on results (maybe a table of metrics with p-values or a graph of lift with CIs), one on recommendation (ship or not) with the business impact estimation. Keep it to ~3-5 slides. This will force you to prioritize what matters (executives don't want all the statistical details, just the conclusions and confidence in them, with any caveats).
    - Practice delivering this presentation. Time yourself for ~5 minutes of explanation. Make sure you can explain any of the methods in simple terms if asked ("what's CUPED?" "why did you do a nonparametric test?" etc.) – since you expect questions. This is akin to an interview scenario where you show a project and they drill into details. Because you've built everything, you will handle this well, but practice will smooth it out.
    - If possible, get a friend or mentor to act as an audience and ask you some questions. Alternatively, record yourself and critique. Common questions might be: "How did you decide on that sample size?", "What would you do if the results were significant on one metric but negative on another?", "How would this framework handle if we had 3 variants instead of 2?" – try to anticipate a few.
- **Deliverables:**
    - The capstone experiment notebook or script, and the resulting presentation (could be added to the repo as PDF or PPTX in a `presentation/` folder).

  - Perhaps write a short reflection in the README or a blog post about the capstone: “We ran a second experiment to validate the framework – results and decision are summarized in slides. This confirms the methods work on different scenarios and demonstrates adaptability.”
  - At this point, everything in the repo should be finalized, so this week also includes making the final push to ensure the repo’s **GitHub issues (see below)** are all resolved.

This 6-week plan is intensive but will make you exceedingly well-prepared. The key is consistency – each week builds on prior, and by the end you haven’t just read about these techniques, you’ve implemented and used them in practice multiple times. That kind of hands-on mastery will come across strongly.

Implementation Tasks as GitHub Issues

To execute the above plan in a trackable way, here are concrete GitHub issues you can create. Each issue corresponds to a chunk of work with clear acceptance criteria and affected files. This helps manage the project and also demonstrates project management skills if someone looks at your repo.

1. **Refactor: Modularize Code into `ab_testing` Package**
2. *Description:* Break up `ab_testing_framework.py` into a package with modules (power, srm, frequentist_tests, etc. as discussed). No new functionality, just restructuring and ensuring all tests still run as before.
3. *Acceptance Criteria:*
   - Source code is organized under `src/ab_testing/` with modules: `power.py`, `srm.py`, `frequentist.py`, `bayesian.py`, `variance_reduction.py`, `sequential.py`, `multiple_testing.py`, `ratio_metrics.py`, `noncompliance.py`, `hte.py`, `diagnostics.py`, `reporting.py`, `simulation.py`.
   - The old script `ab_testing_framework.py` is replaced or refactored to use these modules (or an entry point is created under `ab_testing/__main__.py`).
   - Running the full analysis (via CLI or notebook) produces the same numerical results as before (verify key outputs like p-values, lifts, CI bounds).

4. *Files Touched:* `src/ab_testing_framework.py` (to be deprecated or turned into `src/ab_testing/__main__.py`), new files as listed above. Possibly `README.md` to update usage instructions.

5. **Feature: Power Analysis Functions (power.py)**

6. *Description:* Implement functions for sample size and power calculations for both binary and continuous metrics. Include effect size computations (Cohen’s h, d).
7. *Acceptance Criteria:*
   - `power.required_samples_binary(p_baseline, mde, alpha, power)` returns correct sample per group (validate against known examples or `statsmodels` power). Should handle one or two-tailed via a parameter or by doubling alpha if two-tailed.
   - `power.required_samples_continuous(mu, sigma, mde, alpha, power)` returns correct sample per group for a mean difference, using Cohen’s d or power analysis functions.
   - Include unit tests: e.g. for a baseline 50%, mde 5% (absolute), verify result matches manual calc; for continuous, maybe test a scenario where we expect a certain output (could use `tt_ind_solve_power` from statsmodels as reference in test).

    - Functions have docstrings explaining formula and assumptions.

8. *Files Touched:* `src/ab_testing/power.py`, `tests/test_power.py` (new).

9. **Feature: Randomization Checks (srm.py & diagnostics)**

10. *Description:* Implement SRM chi-square test and baseline balance checks. Possibly integrate these with a higher-level diagnostics function.
11. *Acceptance Criteria:*
    - `srm.check_srm(n_control, n_treatment, alpha=0.001)` returns a p-value for chi-square or a boolean flag. Test this with an obvious SRM case (e.g. 500 vs 600 when expecting equal – should flag p < 0.001). And a no-SRM case (500 vs 500 – p ~1).
    - `diagnostics.baseline_checks(df, cols, treat_col)` runs t-tests (or KS tests) for each column in cols between treat groups, returns any p-values. This is more of a stretch goal; at minimum document that this could be done. If implemented, test on our sim data (should mostly show non-significance since we randomized well).
    - The analysis workflow calls `check_srm` before looking at outcomes, and prints a warning if p < threshold.

12. *Files:* `src/ab_testing/srm.py`, `src/ab_testing/diagnostics.py`, `tests/test_srm.py`.

13. **Feature: Frequentist Test Suite (frequentist.py)**

14. *Description:* Implement z-test for proportions, Welch's t-test for means, and Mann-Whitney U in the `frequentist` module. Possibly also chi-square for contingency tables if needed.
15. *Acceptance Criteria:*
    - `frequentist.z_test_proportions(x_A, n_A, x_B, n_B, two_tailed=True)` returns a dict or object with fields: p-value, z_stat, difference, CI tuple. It should match `statsmodels.stats.proportion.proportions_ztest` output for same input [30]. Test it on at least one scenario (e.g. 50/500 vs 60/500) and compare p-value to known calculation.
    - `frequentist.t_test(values_A, values_B, two_tailed=True, equal_var=False)` returns p-value, t_stat, diff, CI. Should match SciPy Welch's t-test. Test on small samples.
    - `frequentist.mann_whitney_u(values_A, values_B)` returns U statistic, p-value, and rank-biserial correlation. Compare p-value to SciPy's `mannwhitneyu` for same data.
    - The module should handle edge cases (e.g., if all values are identical, maybe return p=1 or a warning). Document any assumptions.

16. *Files:* `src/ab_testing/frequentist.py`, `tests/test_frequentist.py`.

17. **Feature: Bayesian Analysis (bayesian.py)**

18. *Description:* Implement beta-binomial posterior comparison for conversions. Optionally placeholder for continuous metrics (maybe not in-depth).
19. *Acceptance Criteria:*
    - `bayesian.beta_binomial_posteriors(x_A, n_A, x_B, n_B, prior=(1,1))` returns posterior alpha & beta for each group.

    - `bayesian.probability_B_superior(x_A, n_A, x_B, n_B)` uses the above to calculate P(B > A) (via simulation or analytic). Test: in a case where B has much higher conversions, this should be close to 1; if A and B are equal, ~50%. Possibly validate by symmetric cases (if x_A=x_B, result ~0.5).
    - `bayesian.expected_loss(x_A, n_A, x_B, n_B)` to compute expected loss of choosing B if A were true or vice versa (like in the code). Not required but nice.
    - Include unit tests: e.g. if x_A=10, n_A=100, x_B=20, n_B=100, check that P(B>A) is > 95%. And if x_A=x_B, P≈50%.
    - The output of Bayesian analysis in the notebook is consistent with the frequentist result direction (if p was significant, P(B>A) will be high, etc.).

20. *Files:* `src/ab_testing/bayesian.py`, `tests/test_bayesian.py`.

21. **Feature: Variance Reduction Techniques (variance_reduction.py)**

22. *Description:* Implement CUPED and CUPAC adjustments.
23. *Acceptance Criteria:*
    - `variance_reduction.cuped_adjust(Y, X)` returns adjusted Y values (or a tuple of adjusted control & treatment means maybe). Confirm that var(adjusted) <= var(original). You could test on a simple linear relation: X = Y + noise, then adjusted Y should have much lower variance.
    - `variance_reduction.cupac_adjust(Y, X_matrix)` trains a model (use cross_val_predict or internal CV) to get Y_pred, then does similar adjustment. Should return adjusted Y.
    - Both functions documented with how θ is calculated [15]. Possibly provide an API to get θ or model for interpretability.
    - Tests: Create a dummy dataset: Y = 2*X + random noise. Check that cuped_adjust recovers close to 0 variance in Y (since X explains a lot). For cupac, test that if you pass the single X as X_matrix, you get similar result to CUPED.
    - Use these in analysis: show the new SE of difference is smaller. Possibly verify in code by computing SEs.

24. *Files:* `src/ab_testing/variance_reduction.py`, `tests/test_variance_reduction.py`.

25. **Feature: Sequential Testing Utilities (sequential.py)**

26. *Description:* Provide function for O'Brien-Fleming boundaries and sequential decision logic.
27. *Acceptance Criteria:*
    - `sequential.obrien_fleming_boundary(alpha, information_fraction)` returns a z-value cutoff. Test scenarios: if info_fraction=1 (final look), boundary ≈ norm.ppf(1-α/2)=1.96 for α=0.05. If info_fraction is very small (early look), boundary should be much higher (e.g. for 5 looks, first look boundary ~2.8 or so). Possibly cross-check with published OBF tables.
    - If implementing a decision function: `sequential.check_stop(z_stat, current_look, total_looks)` returns True/False for significance at that look. Test with known z_stat and a chosen look.
    - Document that using these boundaries controls type I error ~5%.
    - (Optional) Provide Pocock boundary function for comparison, with tests.

28. *Files:* `src/ab_testing/sequential.py`, `tests/test_sequential.py`.

29. **Feature: Noncompliance & IV Analysis (noncompliance.py)**

30. *Description:* Tools for ITT vs PP vs CACE calculation.
31. *Acceptance Criteria:*
    - `noncompliance.itt_effect(y_control, y_treatment)` simply returns diff in means.
    - `noncompliance.pp_effect(y_control_exposed, y_treatment_exposed)` returns diff in means for those exposed.
    - `noncompliance.compute_cace(effect_itt, take_rate_treatment, take_rate_control)` returns the LATE estimate [10]. Or a version that takes raw data: `estimate_cace(y, treat_assign, treat_received)` to compute via formula (difference in outcome / difference in received).
    - Test: Simulate a simple scenario: 100% of control have no treatment, 50% of treatment get treatment and those who get have +10 outcome. Then ITT effect should be +5, take_rate_treat=0.5, CACE = 5/0.5=10, which matches the actual effect on compliers. Write this simulation in test to verify CACE logic.
    - Include checks to avoid divide by zero (if no one in treat got treated, which is trivial case).
    - Document assumptions (monotonicity, etc.) in docstring.

32. *Files:* `src/ab_testing/noncompliance.py`, `tests/test_noncompliance.py`.

33. **Feature: Heterogeneous Treatment Effects (hte.py)**

34. *Description:* Implement X-Learner or similar CATE estimation and support segment analysis.
35. *Acceptance Criteria:*
    - `hte.x_learner_cate(X, y, treat, model=None)` trains two models (if model not given, use GradientBoostingRegressor as default) and returns an array of CATE estimates for each unit, plus maybe overall ATE. Ensure to do cross-fitting (train $\mu0$ on control, $\mu1$ on treat, impute effects, train τ models).
    - This is complex to unit test fully. Perhaps test on a known synthetic: create two segments with known different effects and check if the average CATE for those segments reflects that ordering.
    - `hte.segment_effect(data, segment_col, outcome_col, treat_col)` computes simple group-wise differences and p-values (like a stratified analysis). Test on simulation data: it should identify the segment with largest effect (Referral, in our sim scenario).
    - Ensure to handle cases where a segment has no treated or no control units (avoid crash, maybe skip or warn).
    - Output of X-learner could be validated by comparing its ATE with the actual ATE (should be close by design).

36. *Files:* `src/ab_testing/hte.py`, `tests/test_hte.py`.

37. **Feature: Experiment Diagnostics & Guardrails (diagnostics.py)**

    - *Description:* Implement guardrail metric evaluation and possibly novelty detection logic.
    - *Acceptance Criteria:*

- `diagnostics.evaluate_guardrails(results_dict, guardrail_thresholds)` takes a dict of metric results (maybe metric -> effect/p-value) and thresholds (metric -> acceptable loss). Returns if any guardrail failed. For example, if guardrail "retention" has threshold -0.02 (no more than 2% drop allowed), and experiment observed -0.01 (p=0.1), it passes; if observed -0.03 (p=0.03), that fails (both statistically and business significance).
- `diagnostics.trend_test(time_series_A, time_series_B)` could be a method to detect novelty effect. Maybe it fits a simple linear model to the difference over time and returns slope significance. Or simpler: compute effect first half vs second half of experiment and see if it drops. If yes, flag potential novelty.
- These can be somewhat subjective; mainly document the approach.
- Unit tests: For guardrails, create a dummy result dict and threshold, ensure it flags correctly. For trend, maybe simulate a decreasing effect and see if it flags a negative slope.
- *Files:* `src/ab_testing/diagnostics.py`, `tests/test_diagnostics.py`.

38. **Enhancement: Reporting Utilities (reporting.py)**

    - *Description:* Functions to format and present results cleanly, and a high-level decision function.
    - *Acceptance Criteria:*
    - `reporting.format_interval(point_est, ci_low, ci_high, unit="pp")` returns a string like "+2.5pp (95% CI: [+1.0, +4.0])". Ensures sign formatting and unit.
    - `reporting.decision_recommendation(primary_result, guardrail_results)` implements the decision matrix logic. For example, primary_result could be a dict with keys `effect` and `p`, guardrail_results a list of dicts. The function returns a string like "Ship", "Do not ship", or "Needs more data", possibly with a rationale.
    - Test format on known numbers (like 0.025, [0.01,0.04]) and check string. Test decision logic with scenarios: (signif positive, no guardrail issue -> Ship; not signif, positive -> Continue; signif negative -> No Ship; guardrail signif negative -> No Ship even if primary positive).
    - Maybe include `reporting.summarize_metrics(results)` that prints a neat table of metrics vs control for multiple metrics (like the one in the output comparing raw vs CUPED effect table). Not critical but nice.
    - *Files:* `src/ab_testing/reporting.py`, `tests/test_reporting.py`.

39. **Documentation & CI**

    - *Description:* Improve documentation, add any missing tests, and set up continuous integration.
    - *Acceptance Criteria:*
    - `README.md` is updated to reflect new structure, with instructions, and an outline of concepts covered. It should also highlight sources used (with citations) and any setup needed to run (like requiring `uv` or one could use pip + pyproject).
    - In-code documentation (docstrings) exists for all public functions, explaining usage and any formulas used.
    - All existing tests pass (`pytest` runs without failures).
    - CI pipeline (GitHub Actions or similar) is configured to run tests on push. This could be as simple as a yaml that installs dependencies (from `pyproject.toml` or `requirements.txt`) and runs `pytest`. Ensure it's passing on the repository.

    - (Optional) Set up `ruff` or `flake8` in CI to enforce style; fix any major style issues. Not strictly needed, but it's a nice touch.
    - *Files:* `README.md`, `CONTRIBUTING.md` (if adding), `.github/workflows/test.yml` for CI, plus many source files if adding docstrings. Possibly `pyproject.toml` if we add dev dependencies for testing.

Once these issues are created, you can tackle them mostly in order (some can be parallel). They cover all enhancements we discussed. Closing each issue will incrementally improve the project until it's fully upgraded.

By addressing all the above, you'll end up with a modular, well-tested, and current repository. It will demonstrate not only knowledge of A/B testing techniques but also code professionalism. Each issue corresponds to a meaningful improvement that could be discussed in an interview ("I refactored the codebase for modularity in issue #1, which taught me about designing reusable functions for experiments").

Throughout the implementation, remember to surface assumptions and failure modes in code comments or documentation. For example, in the sequential test function, note "Assuming independent looks and no information leakage between looks." For CUPED, note "Assumes X is unaffected by treatment and linearly related to outcome." This transparency will further show your depth of understanding and honesty about limitations.

Finally, with the project completed, you'll have a standout portfolio piece. It demonstrates mastery of core concepts, awareness of advanced issues (with citations to prove you studied them), and the ability to implement and communicate them effectively. Good luck, and enjoy the process of building and learning – by the end, you'll be extremely well-prepared for any experimentation-related role or discussion! [3] [5]

---

[1] [5] [6] [7] [27] [46] Novelty effects: Everything you need to know
https://www.statsig.com/blog/novelty-effects

[2] Guardrail metrics: The complete guide to balanced product growth | Signals & Stories
https://mixpanel.com/blog/guardrail-metrics/

[3] [4] [8] [47] [48] How Meta tests products with strong network effects | by Analytics at Meta | Medium
https://medium.com/@AnalyticsAtMeta/how-meta-tests-products-with-strong-network-effects-96003a056c2c

[9] [26] [42] [43] [44] [45] Risk-Aware Product Decisions in A/B Tests with Multiple Metrics | Spotify Engineering
https://engineering.atspotify.com/2024/03/risk-aware-product-decisions-in-a-b-tests-with-multiple-metrics

[10] [11] [12] [23] [24] [25] [32] [33] [38] Encouragement Designs and Instrumental Variables for A/B Testing | Spotify Engineering
https://engineering.atspotify.com/2023/08/encouragement-designs-and-instrumental-variables-for-a-b-testing

[13] [14] [22] [28] [40] [41] alexdeng.github.io
https://alexdeng.github.io/public/files/WSDM2017draft.pdf

[15] [16] [17] [34] [35] [36] [37] Improving Experimental Power through Control Using Predictions as Covariate (CUPAC) - DoorDash
https://careersatdoordash.com/blog/improving-experimental-power-through-control-using-predictions-as-covariate-cupac/

[18] [19] [20] [21] [29] [30] [31] A/B Testing Gone Wrong: How to Avoid Common Mistakes & Misinterpretations | by Suraj Bansal | Medium
https://medium.com/@suraj_bansal/pitfalls-and-mistakes-in-a-b-tests-4d9073e17762

[39] Introducing Differential Impact Detection - Statsig
https://www.statsig.com/blog/differential-impact-detection