ChatGPT

Personalized Learning Engine for MBBS MCQ Exams

Introduction

Medical education in Pakistan's MBBS modular system relies heavily on multiple-choice questions (MCQs) for assessments. A personalized learning engine can significantly enhance students' preparation by adapting to individual strengths, weaknesses, and forgetting patterns. This report presents a comprehensive design for a research-backed adaptive learning system tailored to MBBS students. We will explore core algorithms that track knowledge and adjust practice (Bayesian Knowledge Tracing, spaced repetition decay tuning, ELO ratings, multi-armed bandits, logistic regression, gradient boosting, and Item Response Theory), and show how they can work together synergistically. We also examine successful learning platforms (AMBOSS, UWorld, Anki, NBME) for insights, propose an implementation architecture using FastAPI/Go (backend), React/Next (frontend), PyTorch/TensorFlow (ML), and appropriate databases, and discuss expected performance gains with rigorous testing plans. Throughout, academic findings and real-world metrics are cited to ground the design in evidence.

Core Algorithms for Adaptive Learning

Bayesian Knowledge Tracing (BKT)

How it works: BKT is a foundational model in intelligent tutoring systems that treats student knowledge as a hidden state (usually binary: mastery vs. not) which evolves as the student practices [1]. It is typically formulated as a Hidden Markov Model: each practice opportunity on a skill can result in a correct or incorrect answer, influenced by whether the student knows the skill (with probabilities of a slip if they know it, or a guess if they don't) [1]. Four parameters define a standard BKT model per skill: initial knowledge probability, learning probability (the chance of transitioning from unknown to known after an attempt), slip probability, and guess probability. After each question, Bayes' rule is applied to update the probability that the student has learned the skill, given their answer. This allows the system to "trace" the learner's knowledge state over time and predict future performance [2]. Essentially, if a student answers correctly, the probability they have mastered the skill jumps up (especially if a mistake was unlikely), whereas an incorrect answer lowers the estimated mastery. BKT assumes no forgetting (knowledge state only increases or stays the same), which is reasonable for short-term tutoring on a focused skill.

Applications in education: BKT has been widely used in adaptive learning platforms and intelligent tutoring systems for decades. For example, Carnegie Learning's Cognitive Tutor and various math and science tutors use BKT to decide when a student has probably mastered a skill and can move on [3] [4]. It remains competitive in student modeling – a recent survey noted BKT "offers both competitive predictive accuracy and interpretable parameters" for tracing learning over time [5]. Modern variants integrate Item Response Theory (IRT) concepts or handle multiple skills, but the core idea is prevalent. BKT can dynamically identify a student's weak areas; for instance, if a medical student consistently errs on cardiology questions about heart murmurs, BKT would keep the estimated mastery of that sub-skill low and continue to present

related questions until the probability of mastery crosses a threshold (often 0.95) for long-term retention [6] [7]. In language learning apps and other domains, BKT helps personalize practice by ensuring each learner gets enough practice on each micro-skill (vocabulary word, grammar rule, etc.) until mastery is achieved [8].

Performance metrics: BKT's performance is usually evaluated by how well it predicts the probability of a correct answer on the next practice item for a skill. In educational data mining benchmarks, BKT models typically achieve an AUC (Area Under ROC Curve) around 0.74–0.80 in predicting next-question correctness [6], which is on par with more complex deep learning models for many datasets. For example, one study reported BKT AUC ≈0.74, "in line with BKT's typical performance" on skill mastery prediction tasks [6]. BKT's simple structure sometimes limits its accuracy (it doesn't account for forgetting or multiple subskills without modification), but its interpretability and low data requirement make it attractive. In the context of medical education, where data per student per skill might be limited, BKT's transparency is useful – instructors can see the estimated mastery for each topic and even interpret the slip/guess rates. Performance-wise, a well-tuned BKT could help predict which exam questions a student is likely to get wrong, enabling targeted practice. However, purely BKT-based systems might need augmentation for the spaced-out, long-term learning in medical studies (where forgetting is a factor over weeks/months).

Datasets and benchmarks: Public datasets that have been used to evaluate BKT include the ASSISTments math tutoring dataset (e.g., the 2009 KDD Cup dataset), Cognitive Tutor data (e.g., algebra problems) [9], and more recently the EdNet (RIID) dataset of millions of student interactions (though deep models were more popular there). While these are not medical, they establish baseline expectations for knowledge tracing. In medical education, there is no massive open dataset of student question responses due to privacy and specificity, but a custom system could start collecting its own. We can leverage smaller published studies – for example, a dataset of medical students answering practice questions for USMLE Step exams could be used to fit BKT parameters for various subjects. The **pyBKT** library provides tools to train BKT on any such dataset [10]. Notably, BKT can be fit even on moderately sized data by using expectation-maximization; one benchmark used ~150 students and a few thousand interactions per skill [9]. As a reference point, the Cognitive Tutor dataset had on the order of 1,000 students and a few dozen items per skill [9]. For our MBBS use-case, we might calibrate BKT using historical question bank data (if available from past cohorts) or even simulate some data if needed to initialize the model.

Per-User Decay Tuning (Spaced Repetition and Memory Models)

How it works: Medical curricula cover an enormous amount of content, and long-term retention is crucial. "Per-user decay tuning" refers to modeling each learner's forgetting curve individually, so that the system can schedule review of content at optimal times. In practice, this is achieved through spaced repetition algorithms or memory models that adjust the interval of reviews based on performance. A classic approach is the **spaced repetition system (SRS)** popularized by SuperMemo and Anki, which uses an **ease factor** per flashcard (or per user-item pair) to determine how the review interval grows or shrinks based on whether the user remembered it. For example, Anki's default algorithm (SM2) increases the interval exponentially if the student recalls correctly, but resets or shortens it if they forget, and it maintains a difficulty rating for each card that influences this interval [11]. The key idea is that each user-item has a "decay rate" – some facts the student remembers easily (slow decay, need less frequent review) and others they forget quickly (fast decay, need more frequent review). *Per-user decay tuning* means the system learns those rates from the user's history, rather than using one-size-fits-all schedules.

Modern research-based models like **Half-Life Regression (HLR)** take this further by fitting a regression model to predict the half-life of memory for each flashcard for each user [12] [13]. Duolingo developed HLR to personalize practice of vocabulary: it assumes recall probability decays exponentially with time since last practice (`p = 2^{-Δ/h}` where Δ is time gap and *h* is the item's half-life in that user's memory) [14]. The half-life *h* is predicted by a logistic regression model that uses features of the item and the user's practice history (e.g. how many times it was seen, whether last attempt was correct, etc.) [13]. Each time the user practices, the model updates its estimate of *h*: if the user remembered, the item's half-life increases (meaning next time it can be scheduled further out); if they forgot, the half-life decreases [15]. In formula, HLR might update *h* via new regression weights, or even simpler, Duolingo mentions a conceptual view: "each correct answer increases your memory's half-life, each mistake decreases it" [15]. By modeling this per item and user, the system can decide exactly when you're "on the verge of forgetting" an item and prompt a review at that time [16].

Applications in education: Spaced repetition is extremely relevant in medical education – students often use flashcards to retain vast amounts of information (diseases, drug facts, anatomy details). Anki, a popular open-source flashcard tool, is heavily used by med students, and it implements per-card decay tuning via its "ease" factor. If a student consistently marks a card as "easy," the ease factor rises and future intervals grow larger; if they struggle, the ease drops, scheduling it sooner. This individualized scheduling has been shown to **improve long-term retention** dramatically over cramming [17]. Duolingo's HLR, while used in language learning, demonstrated the power of data-driven decay models: in A/B tests, their personalized scheduling led to a **12% increase in overall user activity** (practice engagement) and a 9.5% boost in daily session retention compared to their old method [18]. Users came back more, presumably because the practice felt more effective and rewarding.

In an adaptive MBBS prep engine, we would use a similar approach: track each question or fact a student encounters, and schedule quizzes on it over increasing intervals to fight the forgetting curve. We could employ a simplified form of HLR or use existing algorithms like SM2, but allow per-user calibration of parameters. For example, some students have better long-term memory and can go longer without review, others need more frequent refreshers – the system can learn this by observing their recall success. Modern variants like the **Forgetting Curve models** in research even incorporate time decay into logistic models (e.g., models that predict probability of recall given time since last seen, prior successes, etc.). A concrete plan is to implement an algorithm such as HLR or a neural network that outputs a predicted recall probability for each Q–student pair over time, and use that to prioritize what a student should review each day. By integrating this with something like Bayesian Knowledge Tracing, we ensure that *both* immediate mastery and long-term retention are addressed (BKT for short-term skill mastery within a module, spaced repetition for keeping mastered content fresh until exam time).

Performance metrics: The success of a decay tuning or spaced repetition algorithm is measured by improvements in recall and retention. Research consistently shows spaced repetition yields large gains: one study on medical students using Anki found those who diligently used spaced repetition scored on average **11.5 percentage points higher** on exams (71.5% vs 60.0% on a standardized test) than those who didn't, a significant difference [19] [20]. Another review noted spaced repetition is "associated with higher success rates in medical school entrance exams" [21]. Duolingo's internal metrics showed their HLR model nearly **halved the prediction error** of memory recall compared to older scheduling methods [22], and as mentioned, user retention improved by 9–12% in live experiments when using the personalized model [18]. We can expect, based on these, that a well-tuned personal decay model in our system would increase students' long-term retention of facts (perhaps doubling the time they remember information before

needing a review) and ultimately improve exam performance. Another metric is **time efficiency**: adaptive scheduling should enable students to achieve the same retention in less total study time. Knewton (an adaptive learning company) reported students using its AI scheduling achieved **62% better test scores** than those who did not [23] – a dramatic result, although it likely combines multiple adaptivity aspects beyond just spacing. For validation, we might use a metric like "percent of content retained after X weeks" as measured by surprise quizzes; we'd expect spaced repetition users to far outperform a control group.

Datasets: There are some public data resources for spaced repetition. Duolingo released a dataset of 13 million user-word practice pairs as part of research on HLR [24], which could be used to test memory models. For medical content, one could use the large **MedMCQA** dataset of medical MCQs (over 194k questions) [25] to simulate a Q&A practice system, though that dataset provides questions and answers rather than student response logs. If we had access to Anki usage data from med students (some studies have gathered aggregate stats), that could inform simulation of forgetting curves. In absence of existing med-specific data, we can apply general memory model findings – human memory decay properties are somewhat universal (following an exponential or power-law forgetting curve). We will incorporate known cognitive parameters (e.g. typical half-life of newly learned medical fact might be on order of days if not reviewed) and then adjust per user based on observed recall.

ELO Rating System for Students and Questions

How it works: The **ELO rating system**, originally developed for ranking chess players, can be repurposed to model student ability and question difficulty dynamically [26] [27]. In an educational context, we treat each question attempt as a "match" between the student and the question: if the student answers correctly, we can think of the student as having "won" against the question; if they get it wrong, the question "wins". ELO assigns a numerical rating to each student (representing their proficiency) and each question (representing its difficulty). After each interaction, the ratings are updated. The size of the update is larger if the outcome was unexpected. For example, if a low-rated student (expected to lose) somehow gets a very hard question right, that's a surprising win – the algorithm will boost the student's rating significantly and/or lower the question's rating (it wasn't as hard as thought or the student learned something). Conversely, if a strong student fails an easy question, their rating drops more. The update rule uses a logistic formula to compute the expected outcome and a constant *K* factor to scale the adjustment. Essentially: `new_rating_student = old_rating_student + K * (Outcome - expected_win_prob)`. If Outcome is 1 (correct answer), the student's rating goes up a bit (especially if expected_win_prob was low); if Outcome is 0 (incorrect), their rating goes down [27]. The question's rating is updated in the opposite direction (because if the student wins, the question "loses" and thus may be easier than its rating suggested). Over time, this process converges so that a student's rating reflects their probability of getting questions right, and a question's rating reflects how often it stumps students of various levels.

Applications in education: ELO-based models have gained popularity because they are *simple, online-updatable,* and require no large batch training [28] [29]. Researchers have noted that ELO is "simple, robust, and effective" for adaptive educational systems, yielding performance comparable to more complex IRT or Bayesian models [29]. It's been used for things like adaptive practice in geography facts [30] and in systems that continuously calibrate question pools. For instance, one can create an adaptive quiz that always selects a question whose ELO difficulty matches the student's current rating – this is similar to how **computerized adaptive testing (CAT)** works, but using ELO rather than traditional IRT. Khan Academy reportedly used an ELO-like system to match students with exercises of appropriate difficulty (though they have since evolved their approach). Duolingo's platform ("Birdbrain AI") also uses an item-response model akin to ELO/IRT to

keep exercises in a "productive struggle" range of difficulty for the learner [31] [32]. In our MBBS engine, an ELO system could be used to **track each student's overall ability or topic-wise ability in real time**, and to continuously adjust question difficulty presented. As the student answers questions correctly, their rating for that subject goes up, and the system gives them harder questions; if they start struggling, the rating falls and the system adapts by giving easier or remedial questions to rebuild confidence. This ensures students are neither bored by too-easy items nor overwhelmed by too-hard ones, echoing Vygotsky's zone of proximal development (keep the challenge level just right).

The ELO system can also help maintain a calibrated question bank: new questions can start with provisional difficulty ratings, and as students attempt them, the ratings adjust. If we have data that a certain anatomy question is often missed even by high-rated students, its difficulty ELO will rise, signaling it's a tough question. Conversely, a question everyone gets right will sink in rating. This dynamic adjustment is useful in a modular medical curriculum where question difficulties might change if curricula change or if some cohorts find certain topics easier.

Performance metrics: ELO can be evaluated by how well the ratings predict outcomes. A good ELO system will yield probabilities of correct answers that match actual performance frequency. ELO's predictive accuracy has been found to be on par with 2-parameter IRT in some studies for large-scale item difficulty estimation [33]. Its advantage is agility and simplicity: *no large student history needed to start*, it updates with each interaction, and inherently accounts for learning to some extent (since ratings change). One study with an adaptive practice system found that using ELO to select content led to improved learning efficiency – students mastered facts in fewer trials because they always got material at the right level [34]. In terms of numbers, ELO's expected error in predicting a given question outcome might be slightly higher than a fine-tuned IRT model early on, but it quickly converges as data accumulates [35] [29]. The cited Pelánek (2016) paper highlights that ELO is "comparably performing" to IRT and BKT while being far easier to implement in an online system [29].

For our context, one metric of success is the stability of student ability estimates – after say 20–30 questions, does the ELO rating for a student stabilize such that we can roughly predict their score on a mock exam? We might simulate an experiment: assign initial ratings and see how quickly the system zeroes in on a known proficiency. Another measure: **match between target and actual difficulty success rate**. If the engine tries to keep a student at ~70% success (optimal challenge), we can see if, after rating adjustment, the student indeed answers ~70% correctly. A well-functioning ELO adaptation should keep each learner in that band (and MDSteps, a new adaptive USMLE platform, explicitly mentions doing this – keeping students in a "productive struggle" zone where they get ~60–80% correct to maximize learning [31] [36]).

Datasets: ELO doesn't require a separate dataset to "train" in the traditional sense, but we can validate it on any Q&A dataset. For example, we could test it on a past set of exam results: take a dataset of students answering questions (like the responses from an NBME practice exam if available, or a simulated dataset), and use ELO updates sequentially to predict performance on later questions. Pelánek (2016) demonstrates ELO on a geography fact drill dataset [30]; Wauters et al. (2011) applied ELO to a small language learning set; those references showed ELO's viability on real student response data. If needed, we could cross-validate ELO difficulty estimates against IRT-calibrated difficulties from an exam like NBME: if there's public data of item difficulties (maybe from literature or released question banks), comparing correlation would be informative. Notably, research found a strong correlation between ELO-derived item difficulty and IRT difficulty (often >0.9 after enough data) [33], meaning ELO can recover the same ordering of item difficulties.

That gives confidence that using ELO online will approximate the gold-standard psychometric difficulty measures.

Multi-Armed Bandit Algorithms for Content Selection

How it works: Multi-armed bandits (MAB) provide a mathematical framework for **sequential decision-making under uncertainty**, balancing exploration and exploitation. In the context of a learning engine, the "arms" could be different types of content or strategies we can show to the student, and the "reward" is a measure of learning gain or engagement from that content. The system must choose what content to present next (which chapter, which question, which resource) to maximize the student's overall learning outcomes. Bandit algorithms (like ε-greedy, UCB, or Thompson Sampling) aim to adaptively try different options and increasingly favor those that yield better results [37] [38]. For example, suppose we have a few alternative paths to teach physiology: one with more diagrams, one purely text, one via clinical vignettes. We might not know initially which works best for a given student. A bandit strategy could try each and observe a reward (say the student's quiz performance or engagement time), then converge to the method that seems to work best for that student. Unlike ELO which focuses on difficulty matching, bandits are great for **content recommendation and strategy optimization**.

A practical use in our system: deciding *which topic or question to give next*. If a student has several weak areas, a bandit algorithm can decide which weakness to address first by estimating the payoff of practicing each and updating those estimates. The reward could be immediate (correct vs incorrect answer, or improvement in BKT mastery, etc.) or longer-term (did practicing topic A lead to better exam scores than practicing topic B?). Contextual bandits incorporate user state as context in making these decisions [37] [39]. For instance, a **contextual bandit** might use the student's current knowledge state and engagement level as context to choose the next question that maximizes probability of learning (reward could be defined as the BKT probability gain, or simply 1 for correct answer if we want to maximize success rate).

Applications in education: Bandit algorithms have been studied for sequencing educational content [40] [37]. Prior work has shown that reinforcement learning and bandits can outperform fixed or random sequencing by tailoring to the student [37]. One research example is a "Diversity-Aware Bandit for Sequencing Educational Content (DABSEC)" which clustered students by learning behavior and used a bandit to choose questions most likely to benefit each cluster, yielding higher rewards (better prediction of responses) than non-adaptive baselines [41]. Another example: a bandit algorithm was used to decide whether a student should get a review question on old material or a new question on a fresh topic at each step – the algorithm learned which strategy led to higher overall retention by experimenting and measuring test outcomes. Bandits are also used in A/B testing of educational features: instead of a standard A/B test (50/50 split), platforms use multi-armed bandits to quickly allocate more users to the better performing variant (this is how some learning apps optimize versions of exercises or hints). For our MBBS engine, we could deploy a bandit to optimize *which learning activity yields the best learning per unit time*. Consider a student weak in both pharmacology and anatomy – a naive scheduler might alternate topics, but maybe the bandit finds that focusing on pharmacology first gives a bigger improvement (perhaps because pharmacology questions also incidentally reinforce physiology knowledge, etc.). The bandit would learn the best order by observing performance gains. Similarly, if we have multiple question sources (easy vs. challenge questions, or text vs video explanations), a bandit can learn which approach leads to higher post-quiz scores for the student.

Roles in the system: In the overall engine, bandits act as the high-level decision policy guiding content recommendation. While BKT/IRT/ELO handle modeling of knowledge and difficulty, the bandit uses those as inputs (context) and tries actions (e.g., "give a hard cardiology question" vs "give a medium anatomy question") and observes reward (did the student's mastery improve? did they answer correctly? was their engagement high?). Over time, it will personalize the learning plan sequence. The result is a *synergistic loop*: the bandit chooses content, the student's response updates BKT/IRT/ELO, which in turn update the context for the bandit's next choice.

Performance metrics: The success of a bandit strategy can be measured in terms of cumulative reward. In learning terms, reward could be defined as knowledge gain or final exam score. One could run simulations or live experiments: e.g., have a bandit-driven sequence vs a static sequence for different student groups and compare learning outcomes (scores, retention). Research from educational data mining indicates that bandit-based sequencing can increase overall learning outcomes – one study noted **higher average reward (better performance)** when using a contextual bandit policy versus a non-adaptive baseline on public educational datasets [42]. If we define reward as "student answered correctly" (a proxy for learning moment success), a bandit algorithm will try to maximize the immediate correctness rate, but we might define it more smartly as "correct on a delayed test of that concept" to truly optimize learning, not just immediate success. Regardless, an effective bandit will quickly identify the best content for the student. We could track metrics like **time to mastery** (does the bandit sequence help students reach mastery in fewer questions?) and **long-term retention** (does it choose an order that results in better retention a week later?). A successful deployment would show improvements in these metrics. For example, an adaptive platform might report that students complete modules 20% faster or retain 15% more information after a month, compared to a fixed curriculum – such data would vindicate the bandit approach.

Datasets: As with other algorithms, we could validate multi-armed bandit logic on existing datasets by simulation. For instance, using the ASSISTments dataset (student responses over time), one can simulate a bandit that picks problem types and see if it would have improved things (though logged data is not experimental, some off-policy evaluation methods exist). Ideally, one would perform live trials in the actual userbase, since bandits shine in *online* settings. There has been a public competition "EDM2010 Challenge" for sequencing and the "Riiid 2020" for knowledge tracing, but not specifically for bandit, although any dataset of sequential interactions (like EdNet) can let you retroactively test: feed a bandit algorithm the sequence and see what it would have done differently and with what predicted reward. In short, we have theoretical backing that bandits help, but the ultimate proof will come from running it with our MBBS students in an A/B test scenario (we'll describe this in the testing section).

Logistic Regression for Student Performance Prediction

How it works: Logistic regression is a simple yet powerful model for predicting binary outcomes (such as correct/incorrect answers) based on features. In personalized learning systems, logistic regression can be used for modeling the probability a student will answer a question correctly, given various features of the student, the question, and their history. A prime example is the **Performance Factor Analysis (PFA)** model, which is essentially a logistic regression knowledge tracing model: it might use features like "number of past successes on this skill" and "number of past failures on this skill" to predict the next answer outcome [35]. The logistic model yields a probability `P(correct) = sigmoid( w0 + w1*successCount + w2*failureCount )`. More generally, we can incorporate many features: time since last practice, difficulty rating of the question, whether hint was used, etc., and train weights to predict correctness. Unlike BKT

(which is a probabilistic generative model) or IRT (which has a specific form), logistic regression makes minimal assumptions and can ingest any available predictors.

Applications in education: Logistic models have been widely used in educational data mining as baseline and interpretable predictors. For instance, PFA (Pavlik et al. 2009) showed that a simple logistic model often outperformed BKT in predicting student responses, because it could separately weight successes and failures and did not assume a binary learned/unlearned state but rather a gradual skill gain. Many learning systems use logistic regression on top of features extracted from student interaction logs – for example, Coursera MOOCs might use logistic regression to predict if a student will drop out or get a quiz right, based on their activity features. In our context, we could use logistic regression in several ways: (1) As a **memory model** (a simpler alternative or complement to HLR) – e.g., predict recall probability with features like time since last review, number of prior reviews, etc. (2) As a **student modeling tool** – predict exam pass probability or question correctness given student's concept mastery features. (3) As part of the **IRT/BKT hybrid** – indeed, a 2-parameter IRT model *is* a logistic regression where features are (ability – difficulty) for a question; logistic regression can generalize that by adding more features (discrimination, guess factors, or even content tags).

One concrete application: we can train a logistic regression to predict the likelihood a student will get any given MCQ correct, using features such as the student's overall MBBS phase (year of study), their BKT mastery for that topic, the question's difficulty rating, how many similar questions they've seen, whether they saw an explanation for that topic before, etc. This model could then be used to choose questions (perhaps by targeting the 50% probability sweet spot for learning challenge or to estimate how much a question will stretch the student). Another use is to identify at-risk students: logistic regression on historical data might show that certain patterns (like low practice time and low scores in certain core topics) predict failing the final exam – the engine can then intervene or alert instructors.

Performance metrics: As a predictive model, logistic regression is usually measured by AUC or accuracy in classification tasks. In knowledge tracing, logistic models often achieve AUC in the 0.75–0.85 range on next-question prediction (depending on feature richness and dataset complexity). In the 2010 KDD Cup (educational challenge), simpler logistic/PFA models had competitive results close to BKT. In the 2020 Riiid Answer Correctness Prediction competition (a large-scale knowledge tracing challenge), a tuned **XGBoost (a GBM) or logistic** model was able to reach an AUC around 0.78–0.79 on the validation data, whereas the best deep learning hit ~0.82 [43] [44]. This shows that with enough data and features, even generalized linear models can do quite well. Logistic regression's coefficients are interpretable; for example, we could see that "each past failure on a topic reduces odds of next success by factor X" or "an extra day of gap reduces odds by Y%", which is valuable insight for educators.

Datasets: We can train logistic models on any historical student response dataset. If we accumulate data from students using our platform (thousands of interactions), a logistic model can be fit to predict correctness. Public datasets to experiment with include the ASSISTments dataset (which includes features like attempts and performance by skill) – one can create logistic models per skill from that. Another is the EdNet (RIID) which has ~100 million interactions; one Kaggle solution shows using logistic features as input to a model [45]. For medical-specific data, we might rely on smaller datasets: for example, if we have a set of students who all took a bank of 200 practice questions and then their actual exam result, a logistic regression could be trained to predict pass/fail from their practice question performance pattern. That could validate the model's utility in identifying who needs more help. Even without explicit med datasets, we can leverage literature: e.g., a logistic model might incorporate "number of Anki cards reviewed" as a

feature – a study found number of mature Anki cards was a significant positive predictor of exam score (β ≈ 0.65, p=0.026) [46] [47], meaning more flashcard practice leads to higher scores. We could include such features to our models to predict outcomes and adapt accordingly (e.g., recommending more flashcard work if predicted score is low).

Gradient-Boosted Machines (GBMs) for Prediction

How it works: Gradient-boosted machine algorithms (like XGBoost, LightGBM, CatBoost) are powerful ensemble methods that build decision trees sequentially to minimize prediction error. They can handle large numbers of features, capture non-linear interactions, and often achieve top performance on structured data tasks. In our engine, GBMs can be used similarly to logistic regression but can exploit more complex patterns. For example, a GBM could take the raw event log of a student (response history, timing, etc.) and predict something like "probability of answering the next cardiology question correctly" or "expected score on the upcoming exam". The boosting process will combine many weak predictors (trees that might pick up one feature threshold like "if mastery > 0.8 and time spent < 30s, predict correct") into a strong predictor. GBMs often yield higher accuracy than a single logistic regression because they can model feature interactions (e.g., the effect of time gap might be different depending on number of prior reviews – a tree can capture such interaction, whereas a basic logistic model might need manual feature crosses).

Applications in education: GBMs have been used in analysis of educational data for things like dropout prediction, grade prediction, or even adaptive learning contest solutions. In the Riiid AI Education Challenge (knowledge tracing), many top participants ensembled GBMs with neural networks [48], and some purely tree-based approaches scored competitively due to the huge data and many engineered features. In a deployed system, one might not retrain a GBM in real-time for each student (since GBMs are more heavyweight than ELO or logistic), but one could periodically train GBMs on accumulated data to discover patterns and inform the other models. For example, a GBM could be trained to predict which question concepts are most likely to lead to learning gains for certain students, essentially informing a policy. Or a GBM could predict a student's final exam score given their current practice statistics – if the prediction is below a threshold, the system can then alter the study plan (maybe suggest more questions in weak areas or more review). Another use: optimizing content difficulty – a GBM could analyze item text/features to predict difficulty level (like using NLP features of questions to predict their IRT difficulty, helping to tag new questions without needing a full calibration test).

Performance metrics: GBMs generally excel in predictive accuracy on tabular data. If we measure by AUC or RMSE on predicting student performance, a well-tuned GBM with enough data can often exceed simpler models by a few points. For instance, in one knowledge tracing dataset, a logistic model might get AUC 0.75, while a GBM could reach 0.80 by capturing non-linear effects and combining multiple skills data. The trade-off is interpretability (trees can be interpreted to a degree, but hundreds of trees less so than a few logistic coefficients) and the risk of overfitting if not enough data. However, feature importance from a GBM can tell us useful things: e.g., it might reveal that "time since last attempt" is the most important predictor of an error, or that "previous year's exam score" is the strongest predictor of current performance, etc. These insights can refine our model choices (like emphasize spaced repetition if time gap matters a lot).

In terms of boosting learning outcomes, using GBM predictions in the loop can help target interventions more effectively. If a GBM predicts a student is 60% likely to pass an upcoming module exam, the system can initiate a proactive review schedule or notify mentors. The success metric for using GBMs would be an improved ability to identify struggling students early and personalize their plan to ultimately raise that

outcome. We might measure something like the **precision/recall of identifying at-risk students** or the calibration of predicted scores vs actual scores. If the GBM is well-calibrated, a student predicted 80% likely to pass should indeed have ~80% chance; any large calibration errors would mean our model is misreading the data.

Datasets: Training GBMs requires a dataset of instances (which could be student-question interactions or student exam outcomes). If our system is new, we might start with a combination of synthetic data and data from literature to pre-train models. For example, we have studies linking Anki usage to scores [19], or time spent to performance – we can incorporate those relationships. As data comes in, we retrain. Public data like the *EdNet* (which includes question attempts with timestamps, etc.) can be a playground to develop the GBM approach. Kaggle hosts the **Riiid Answer Correctness Prediction** dataset with over 1M students and 100M interactions; one can train a GBM on a subset to predict correctness and see feature importances. While that's not med-specific, some features like "explanation viewed" are analogous to "rationale read" in a medical question context. We could also look at the NBME's recent open datasets (there was a 2022 Kaggle competition by NBME about extracting information from clinical notes; not directly relevant, but NBME also sometimes publishes correlation studies). Ultimately, the most relevant data will come from usage of our platform once launched, but leveraging open academic data to inform initial model development is wise.

Item Response Theory (2PL/3PL IRT)

How it works: Item Response Theory (IRT) is a family of models from psychometrics that relate the probability of a correct answer to latent traits like student ability and item difficulty. In the classic 2-Parameter Logistic (2PL) IRT model, the probability that student *j* answers item *i* correctly is given by: `P_{ij} = 1 / (1 + exp(-a_i*(θ_j - b_i)))`, where θ_j is the student's ability, b_i is the item's difficulty, and a_i is the item's discrimination parameter (how sharply it differentiates high vs low ability) [49]. The 3PL model adds a guessing parameter c_i to account for a non-zero floor probability of guessing correctly on multiple-choice items. In 3PL, `P_{ij} = c_i + (1 - c_i) / (1 + exp(-a_i*(θ_j - b_i)))`. Intuitively, if the student's ability θ is equal to the question's difficulty b, the chance of correct is 50% (assuming no guess factor). If ability is much higher than difficulty, the probability approaches 1 (they should get it right easily); if much lower, the probability is near 0 (beyond their current level). The discrimination a_i controls the slope – high a means the probability curve is steep (the item really separates high vs low ability cleanly), whereas low a means even high ability students might have some chance to slip or low ability might sometimes manage it, making the item less decisive.

To use IRT, one typically needs to calibrate item parameters (a, b, c for each question) using past exam data or pilot testing. Once calibrated, we can estimate a student's ability by their pattern of answers (often via maximum likelihood or Bayesian estimation). IRT is widely used in standardized testing (NBME exams, USMLE, etc.) to score examinees and equate exams – the ability score can be mapped to a scale (like USMLE's 3-digit score) and difficulty parameters ensure fairness across different test forms.

Applications in education: In an adaptive learning engine, IRT can serve multiple roles. First, it provides **content calibration** – if we have IRT parameters for each question in our bank, we know which questions are easy or hard (b parameter) and how reliable they are (a parameter). This can guide content selection (e.g., if a student's estimated ability is θ, we might choose items with difficulty around θ to challenge them appropriately, similar to how CAT selects items to maximize information). This approach would ensure, say, a student who has intermediate ability in pharmacology gets medium-hard pharmacology questions, rather than wasting time on very easy ones or being lost on ultra-hard ones [32]. Second, IRT gives a principled way

to **aggregate performance into an ability score** that can be tracked. Instead of saying "student got 80% correct", IRT would quantify ability while accounting for question difficulty – 80% correct on easy questions yields a lower θ than 80% on hard ones. This is important if we want to accurately predict how a student would do on an actual MBBS exam (which can be thought of as an IRT-scored test itself).

Many modern learning systems use IRT in the background. For example, Duolingo discussed that their BKT's learned parameters resemble logistic IRT parameters [49] – essentially acknowledging the link between knowledge tracing and IRT (one can think of mastering a skill in BKT as having ability > difficulty for that skill's questions). NBME likely uses IRT to analyze questions in their question bank and to score practice assessments. If our engine could integrate with NBME's scale (for instance, estimate a student's NBME subject exam score by treating our question responses with IRT parameters anchored to NBME standards), that would be very valuable to learners.

Performance metrics: IRT models are typically evaluated by **goodness of fit** to data (likelihood or information criteria) and by the reliability of ability estimates. A well-calibrated IRT model will be able to predict the probability of each question being answered correctly with high accuracy. One can measure an analog of AUC or RMSE for IRT predictions on a validation set. Because IRT assumes static ability, pure IRT doesn't capture learning – but in a testing scenario it's very powerful. If we do periodic assessment, IRT can measure growth in ability over time (although strictly speaking that violates the static assumption, we can apply IRT at each snapshot). The **information function** in IRT tells us how much measurement precision each item gives at different ability levels; using those, we can quantify that our adaptive system's questions are providing good coverage. For example, we might want to ensure that for each topic, we have a range of difficulties such that both weak and strong students can be accurately assessed. If we find that all our cardiology questions are very hard (b high) and a student is low ability, we'll get many wrong answers but not much discrimination between "kinda weak" and "very weak" – IRT would flag that as providing low info at the lower θ range. So one performance metric is the **test information curve** – ideally, our item pool and our adaptive selection produce high information (i.e., low standard error in ability estimate) across the range of abilities of our students [50]. In practice, if our engine can estimate a student's ability with a standard error of, say, 0.3 logits after 50 questions, that's good; if it's 0.1, that's excellent (meaning very precise).

Another evaluation is **correlation of our estimated abilities with real outcomes**. If we estimate a certain θ for a student, and later they take an actual exam, does a higher θ correlate strongly with higher exam score? That would validate our engine. Often, IRT-based scores correlate well with criteria because they are essentially what standardized tests use. We could strive for, e.g., a Pearson correlation >0.7 between our internal ability estimate and students' university exam results – that would demonstrate the engine's assessments are meaningful.

Datasets: To calibrate IRT in the medical domain, one usually needs a sizeable dataset of student responses to a set of questions. If we have historical data (e.g., past classes taking the same test questions), we could run IRT calibration (using software like pyirt, scikit-learn's logistic, or custom PyTorch models). In absence of that, we can borrow parameters from literature: there is a dataset called **MedMCQA** with entrance exam questions; while it doesn't have student performance data, it's plausible the exam board had difficulty stats for those questions. Another approach: use our initial user base to perform calibration – the engine might begin with a trial phase where it treats questions as experimental and uses student results to estimate b and a. Even a few hundred responses per question can give a rough estimate. We could also integrate with known difficulty sources: for example, NBME sometimes publishes "p-values" (proportion correct) for

released questions, which can be converted to rough difficulties (p = 0.7 corresponds to easier, maybe b around -0.5 in logistic scale; p = 0.3 corresponds to b around +0.5, etc.). We might seed our bank with difficulty estimates derived from expert judgment or historical passing rates, then refine via IRT as data comes in.

It's worth noting that IRT and ELO are related; in fact, an online ELO update tends to approximate a 1PL IRT (Rasch) model learning over time [51]. We might use ELO early on for convenience, and once enough data is gathered, perform a full IRT analysis offline to solidify the parameters and then switch to an IRT-based CAT approach for fine-grained accuracy [50]. The combination of BKT (for dynamic mastery within modules) and IRT (for cross-module ability measurement) can give a comprehensive picture of a learner's progress [50].

Synergistic Use of Algorithms in a Personalized System

No single algorithm suffices to optimize learning; the strength lies in **combining these techniques** so they complement each other's roles. Here we design a cohesive engine where each algorithm handles the aspect it's best at, and data flows between them to inform decisions. **Figure 1** illustrates the high-level flow of information and decisions in the system (from student interaction to knowledge modeling to content recommendation and back):

Figure 1: Schematic of the adaptive learning engine architecture. Learner interactions (answers, time, feedback) feed into knowledge tracing and memory models, which update the learner's profile (mastery levels, difficulty ratings, forgetting curves). A personalization engine then decides the next content (question or resource) to present, using multi-armed bandit logic guided by the learner model and content metadata. The system continuously loops, with data aggregated for analysis and model updates.

Roles of each algorithm: Each component will take on specialized tasks in the learning plan:

- **Bayesian Knowledge Tracing (BKT)**: This will maintain an up-to-date estimate of the student's mastery for each *fine-grained skill or concept* in the MBBS curriculum. For example, within Physiology, BKT might track "Cardiac cycle understanding" as a skill; within Pathology, "Mechanisms of anemia" as another. Every time the student answers a question tagged with a concept, the BKT model updates the probability that the student has mastered that concept. BKT's binary mastery interpretation is useful for *prerequisite structure*: if certain foundational concepts aren't mastered, the system knows to address those before moving to more advanced ones [8]. BKT will thus drive the **assessment of learning gaps** – if a concept's mastery remains low despite practice, that concept is flagged for more practice or remedial resources. BKT updates happen in real-time with each question attempt (since the computations are quick, essentially a few multiplications for Bayes' rule). The output of BKT (the mastery probabilities) flows into other components: for example, the bandit or recommendation system can use these probabilities as context (e.g., focus on low-mastery areas). Also, BKT's estimated mastery can be used to decide when to consider a topic "learned" and move on (e.g., once mastery > 0.9, that topic's questions could be seen less frequently, except for spaced review).

- **Per-User Decay/Spaced Repetition Model**: This can be thought of as the long-term memory tracker. It works at the level of specific items or facts. After BKT deems a concept mastered *now*, the decay model ensures that the concept will be revisited later to retain it. Concretely, our system will maintain for each student-item pair a next review time or a memory strength. We might implement

this via a Half-Life Regression model or an algorithm akin to Anki's, which updates an "ease factor" and schedule after each review [11]. The memory model's role is to decide *when* to resurface a question or a topic. For example, after a student correctly answers an anatomy question about cranial nerves, BKT might mark that skill as mastered, and the memory model schedules a revival of that or similar question in, say, 3 days. If they still get it right then, schedule next in 2 weeks, and so on, tuning intervals to the student's performance. This ensures *persistent learning*: the knowledge state from BKT is not just one-and-done, but maintained until exam time. The spaced repetition engine will interface with a scheduling queue – essentially, each day the system can pull from the queue any items due for review. Those review questions are then fed into the learning sessions. If a student ignores reviews, the model can adjust (maybe lower their strength). The per-user decay model thus feeds into content selection (ensuring older material due for practice gets priority among the "what next" options).

- **ELO Rating System (dynamic difficulty)**: We will maintain an ELO rating for each student on each major subject or overall, and an ELO difficulty for each question. This gives a quick gauge of a student's level and question hardness, continuously updated. The primary use of ELO will be **real-time difficulty adjustment**. For instance, suppose a student's "Pharmacology ELO" is 1200 and a particular question's ELO difficulty is 1300 – the expected probability of correct is around 0.36 (if we map ELO difference to expected score via logistic). That might be okay if we want to challenge them, but perhaps we aim for ~0.6 difficulty. The system can search for a question with ELO near 1150–1200 to match the student. ELO updates after each question will refine this matching [27]. This mechanism works hand in hand with BKT: BKT might say "student is weak in Topic X", and ELO can say "within Topic X, what difficulty level question should we give?" So, the flow might be: choose a concept via BKT/bandit (say Topic X), then use ELO to pick a question from Topic X that fits the student's current ability (not too easy/hard). ELO also helps us **calibrate new questions** – early attempts will quickly adjust a new question's rating. Additionally, the student's aggregate ELO can serve as an *estimated ability level* that can be communicated (like a progress score) and used as context in bandit decisions (the bandit might choose differently for a high-ELO student vs a low-ELO student, e.g., offering extension material vs remedial basics).

- **Multi-Armed Bandit (MAB) for content sequencing**: The bandit is essentially the decision-making policy engine that *decides what content or action maximizes learning at each step*. We can formalize arms as different *categories of actions*. For example, one arm could be "give a new question in the weakest current topic", another arm "give a review question from the past", another "present a summary or mnemonic", or even non-question activities like "show a quick video on a troublesome concept". The reward for the bandit needs to correlate with learning. We might define an immediate reward such as +1 for a correct answer (encouraging bandit to help student get things right) and maybe an additional bonus for improvement in BKT mastery or for long-term retention signs (this can be tricky to design; a simpler proxy might be just using correctness and perhaps time-on-task or confidence ratings). The bandit (possibly a contextual Thompson Sampling or UCB) will use student state (mastery levels, recent fatigue level, etc.) as context [39] to select the next action. Over time, it learns which sequences yield the best outcome. For instance, it might learn that after 3 wrong answers in a row (context: frustration high), the best action is to switch to an easier question or a different topic to avoid discouragement (reward in terms of getting the student back on track). Or it might learn that a pattern of alternating subjects yields better retention than sticking too long on one subject (reflected in more questions correct later). The bandit essentially ties everything

together by *orchestrating the study plan*. It explores new strategies occasionally (to discover what works if unsure) but mostly exploits the known best strategy for the individual.

- **Logistic Regression / GBM (analytics and prediction)**: These models will run more in the background, on the data aggregated from all users (or all sessions of a user) rather than in the immediate per-question loop. They can be retrained periodically (e.g., nightly or weekly) to analyze trends and make predictions that inform adjustments to the engine. For example, we could have a logistic/GBM model predict "Will the student answer the next question correctly?" using detailed features. If this model's prediction is sharply lower than the BKT or ELO prediction, it could indicate something – maybe the student is fatigued or some feature (like time of day or recent streak) is affecting performance. The system could then adapt (perhaps the scheduling decides to offer a break or easier content). Another model might predict "Exam score given current progress" – if it predicts a student is likely to score 60% on the upcoming exam, and the passing is 70%, we know extra intervention is needed. These predictive models essentially provide a *second layer of adaptivity*: they look at the big picture and can trigger macro-level changes (e.g., recommending the student to revisit foundational topics, or engaging a human tutor) that the micro-level bandit might not handle because it's focused on immediate next step. Furthermore, these models can evaluate the effectiveness of strategies: by including features representing algorithms used (e.g., did the student use the flashcard mode? Did they watch videos?), the GBM could tell which features most strongly predict improvement, guiding us to refine the system (similar to how educational researchers measure what factors correlate with success). In short, logistic/GBM models serve as the **analytics brain**, learning from data to refine parameters and suggest changes to the other components. They won't directly choose content each minute, but their output might periodically update the bandit's policy or the ELO K-factor or identify new feature to feed into BKT (for instance, if GBM finds time-spent is a big factor, we might incorporate time-decay into BKT or slip probabilities).

- **IRT (2PL/3PL) for assessment and difficulty calibration**: IRT will be used mainly in two places: (1) *Initial calibration* of the question bank and periodic re-calibration. *Using accumulated data, offline IRT analysis will refine each question's difficulty (b) and discrimination (a). This can improve the ELO system (we might even initialize ELO ratings based on IRT b estimates). High discrimination questions (a high) are especially valuable for assessment, so the system might mark those as good exam practice questions. If some questions have very low discrimination (maybe poorly worded questions that even strong students miss or weak students get right by guessing), IRT will highlight that and we might retire or revise those questions* [52] [53]*. (2)* Personalized testing. *The engine can generate mock exams or quizzes adaptively using IRT: by continuously estimating the student's θ (ability) on a scale, it can select questions that maximize information for that θ* [50]*. This is basically a computerized adaptive test (CAT) methodology. Over, say, 20 questions, the system can zero in on the student's ability estimate with an error margin. This could be offered as a feature: e.g., a student can take a 20-question adaptive quiz in Medicine and get an ability report that correlates to expected percentile. IRT ensures that this short test is as accurate as a longer static test because it picked the most informative questions for that student. So IRT works hand-in-glove with ELO in daily practice (since ELO is a quick proxy for matching difficulty), but for any formal progress test or prediction of exam score, we'd lean on IRT for rigor. The data from BKT (mastery) and memory (retention) can also flow into IRT analysis by helping define the latent ability at different times. Indeed, there are hybrid models** (sometimes called dynamic IRT or latent trait tracking) that adjust θ as a function of practice – akin to treating learning as increase in θ over time. In our synergy, we might assume θ can grow as the student learns, and we could use the history of BKT mastery gains

as evidence of θ increase. Essentially, at any snapshot, IRT gives a static ability measure, while BKT tells us the trajectory of mastery; combining, we get both current level and momentum of learning.

Data flow between components: The system operates in a loop each study session:

1. **Student Action → Data Capture**: The student attempts a question (or other activity) on the frontend. The system records the result (correct/incorrect), time taken, any hint usage, confidence rating (if we ask the student), etc. This data point is immediately sent to the backend.

2. **Knowledge Model Update (BKT/ELO/Memory)**: The backend's learning model service (likely in Python) receives the interaction. First, it identifies which concept or skill the question was tagged with and updates the BKT model for that concept [2]. For example, if the question was about renal physiology, the "renal physiology" skill's mastery probability is updated given the outcome (rising if correct, dipping if incorrect). Simultaneously, the ELO ratings update: the student's rating for "Physiology" is adjusted and the question's difficulty rating is adjusted accordingly [54] [29]. If we maintain sub-category ELOs (like separate for cardio, renal, etc.), those update too. The memory model checks if this question was in the review schedule – if so, update its next due date based on success/failure; if it was a new question, it might initialize a record for spaced repetition (e.g., schedule a review in a few days if it was learned). All these updates are fast and happen in memory/with minimal DB calls (we might cache the user's state in Redis for speed, so updating BKT/ELO just means modifying the Redis entry for that user's state vector).

3. **Contextual Data Formation**: After updating, the system compiles the current context for decision: e.g., *Student 123* now has mastery: Cardio 0.95, Renal 0.6 (just improved from 0.5), etc.; ELO: Physiology 1300 ±50, Anatomy 1200 ±50, etc.; maybe "learning fatigue score" from time spent or last 10 answers (if 9/10 were wrong, maybe frustration is high). This context vector is fed to the personalization engine (bandit and/or a rules engine).

4. **Decision Engine (Bandit + Rules)**: The multi-armed bandit, considering this context, computes the expected reward of each possible next action [37]. Suppose arms are defined as: A1 = "Next question in weakest mastery area" (exploit weakness), A2 = "Review a past topic due for review" (reinforce memory), A3 = "New question in current topic" (if currently focusing a module), A4 = "Show learning resource (no question)" (if frustrated), etc. The bandit might use an algorithm like Thompson Sampling: it samples a guess of which arm is best based on learned probabilities of reward. If the student has low renal mastery and the bandit's learned model says "addressing weaknesses yields +0.2 reward typically" versus reviewing yields +0.1, it will pick the weakness-fixing arm. Additionally, there could be some rule-based overrides: e.g., if the student got many wrong in a row (sign of frustration), we might override a purely difficulty-based decision to give an easier question or a short break activity to reset morale – these rules can be integrated as context features or as safety constraints on the bandit. In general, the bandit chooses *which concept and type of activity* to do next. Let's say it picks: "focus on Renal Physiology (weak area) with a new question".

5. **Content Selection (IRT/ELO guided)**: Given the bandit's high-level choice (Renal question, new), the system next selects a specific question from the question bank. This is where item models kick in: we have a pool of Renal questions of varying difficulties. The student's current estimated ability in Renal (from ELO or IRT) is considered. We likely want a question that is challenging but doable – perhaps target ~70% probability of success. If the student's Renal ELO is 1100, we pick a question with

difficulty ELO near that. If we have IRT b parameters, we might choose an item with b slightly above the student's θ (to stretch them a bit). We also ensure it's not something seen too recently unless it's meant as a review. The content database (likely in PostgreSQL, with fields for topic, difficulty, last seen, etc.) is queried with these criteria: "Renal Physiology questions, not answered in last 1 month, difficulty ~X, not too similar to last question, etc.". The query could also incorporate **Neo4j** if we want to follow a concept graph – say the bandit chose a general area, we then use the knowledge graph to pick a specific node that's a prerequisite or related concept the student needs. For instance, within Renal, if the knowledge graph shows the student is weak specifically in "Renal autoregulation", we pick a question tagged to that node. Neo4j can help by storing relationships, so if a student struggled with a question on X, maybe the next question is on a prerequisite of X.

6. **Deliver Content and Iterate**: The chosen question is delivered to the front-end for the student to attempt, along with any auxiliary info (like if it's a review question, maybe marked as such). This completes one loop cycle. The student's attempt on this new question will generate the next data point and we repeat steps 1–5. This continuous loop ensures an individualized path: one student might be doing a lot of reviews if their memory model indicates forgetting, another might be rapidly progressing to new, harder questions if they keep mastering content easily.

Diagrams and data flow: The above process was depicted in Figure 1. To summarize the interplay: **BKT and Memory models feed the bandit with knowledge state**, **ELO/IRT feed the bandit with difficulty/ability matching**, **the bandit chooses an action which is executed by selecting content via the item models**. After the action's result, all models update: BKT mastery may increase (leading the bandit to shift focus next time), ELO might increase (leading to harder next items), memory model might push the next review of that item further out. The synergy is such that *short-term learning (BKT) and long-term retention (spaced repetition) are both optimized, while keeping students appropriately challenged (ELO/IRT) and engaged (bandit tries different content types as needed).*

To illustrate with an example: A student is doing microbiology questions. BKT shows they are weak in bacteriology but okay in virology. The bandit, seeing low mastery in bacteriology and that a review for an old immunology question is also due now (per memory model), has to choose: Should it drill more bacteriology now, or interject the immunology review? Perhaps the bandit has learned that switching topics too often reduces learning efficiency for this student, so it continues with bacteriology until some mastery threshold is met. It picks a bacteriology question. ELO sees the student's bacteriology rating is, say, low, so it picks an easy-medium difficulty question to not discourage them initially. The student gets it correct. BKT mastery for "Staphylococcus aureus infections" goes up from 0.4 to 0.65. ELO raises their rating a bit. Memory model schedules that specific question for review in a week. Now, BKT still shows bacteriology not mastered (still <0.8 overall), so the bandit again chooses another bacteriology question. This time maybe a bit harder (ELO suggests bumping difficulty). The student gets it wrong. BKT goes down a bit for that concept, ELO lowers rating. The bandit might interpret two things: (a) the concept is still not grasped – continue – but (b) the student may be getting frustrated after a wrong attempt. If multiple wrongs happen, a rule may fire to switch context (maybe do a quick review of basics or an easier question to rebuild confidence). Meanwhile, the memory model's scheduled immunology review is still pending; eventually, after say 5 bacteriology questions, the bandit might decide to do that review to reinforce old knowledge (especially if it sees diminishing returns on bacteriology attempts at the moment). In that review question, the student's memory model for immunology updates. Over a full study session, the student ends up covering new material (bacteriology) and reinforcing some old (immunology) in an optimal mix. At the end, the engine can output an update: "You improved your mastery in Bacteriology from 20% to 50% today, and

maintained your Immunology mastery at 80%. We'll revisit Bacteriology tomorrow to push it further, and also review Pharmacology which is coming due." This kind of synergy is only possible because the various algorithms share information (mastery percentages from BKT, difficulty from ELO, due times from memory, etc.).

Insights from Existing Platforms and Projects

To ensure our design is grounded in proven strategies, we examine how existing medical education platforms and open-source projects implement adaptivity and what outcomes they report.

AMBOSS

Adaptivity approach: AMBOSS is a medical learning platform that combines a QBank with an extensive knowledge library. AMBOSS markets itself as having *personalized, adaptive learning* features: it dynamically links content and questions, integrates spaced repetition, and provides personalized study recommendations [55]. From user and institutional perspectives, AMBOSS can identify a student's weaknesses and suggest reading materials or questions accordingly. For example, if a student gets a cardiology question wrong, AMBOSS will link to the article on that topic for review, and those topics might be marked for the student to revisit. It also has a feature where within its articles, you can quiz yourself – likely adapting to what you got wrong. Notably, AMBOSS has an **Anki add-on integration**, which means it supports spaced repetition: a student's Anki flashcards can link to AMBOSS content for reference, and presumably AMBOSS's own platform might schedule quizzes (though specifics are not fully public). AMBOSS also mentions "dynamic interlinking" – this suggests that as you answer questions, the system highlights related topics you should study, essentially guiding you on a personalized path through their content library [55].

Measurable outcomes: While AMBOSS doesn't publish A/B test results publicly, its value is reflected in user adoption and feedback. Students often report that using AMBOSS's recommended readings and analysis of their performance helps them cover gaps more efficiently. For instance, AMBOSS provides analytics like "you are 60% prepared in Gastroenterology" based on question performance, and then it tailors recommendations – which is similar to our BKT mastery concept. We can glean that institutions that deployed AMBOSS saw improvements: one case study (on Amboss's site) likely mentions students improving their scores after using it, but quantitative data is sparse. However, adaptive elements in AMBOSS presumably contribute to students being able to focus on weak areas, which should yield higher exam scores or fewer blind spots. As an anecdote, a user might say "AMBOSS's personal recommendations saved me time by focusing on my weak points, and I ended up scoring above average in those topics."

Interface/design ideas: AMBOSS's interface seamlessly blends reference and questions. It might show a student a difficulty meter or use color-coding (e.g., a red highlight on topics you frequently miss). Our design can emulate this by presenting concept mastery dashboards and linking directly to explanatory content in areas of weakness. Another design from AMBOSS is the *Attending Mode*, which can hide the answer explanations initially to encourage active recall – a feature to consider to prevent students from passively reading solutions. Also, AMBOSS's library integration suggests having an on-hand repository of explanations for every question or topic, which our system should as well (perhaps linking to a digital textbook or Wikipedia for medical content).

UWorld

Adaptivity approach: UWorld is one of the most popular exam prep question banks (especially for USMLE). Historically, UWorld's model is not heavily adaptive in terms of algorithm-driven sequencing – it's more user-controlled. Students typically select questions by topic or random sets, and UWorld provides detailed explanations for each question. However, UWorld does have some adaptive features: it provides performance feedback and analysis by subject/systems, and a self-assessment that yields an estimated score. UWorld's strength is in its **content quality and analytics** rather than sophisticated algorithmic personalization. On the analytics side, it shows users the percentage correct in each subject, how they rank compared to others, and it highlights *weak areas*. This nudges students to do more questions in those weak areas (manual adaptivity). There was mention of a "smart algorithm to show your weaknesses" on forums; one Reddit discussion asked if UWorld has an algorithm for weaknesses and the consensus was that UWorld mostly leaves it to the user, unlike something like Kaplan's QBank which had a tutor mode [56]. UWorld's focus is realism and depth: it often recommends going through all questions regardless of adaptivity, because each question teaches a concept. In that sense, UWorld ensures coverage but not necessarily optimized sequence.

Outcomes: UWorld's effectiveness is evidenced by the fact that a huge majority of students who score high on USMLE have thoroughly used UWorld. One reason is the detailed explanations and the fact that questions are challenging (so by grappling with them, students learn). From an adaptive engine standpoint, UWorld demonstrates that *quality of content* is paramount – even if you have the best algorithms, if the questions and explanations are subpar, learning suffers. So our engine must invest in quality explanations and evidence-based questions. UWorld does have two self-assessment exams that predict your actual exam score; these are likely calibrated with an IRT-type method behind the scenes (since they can predict scores within a small margin). The **measurable outcome** here is predictive validity: UWorld self-assessments correlate with actual USMLE Step scores with an correlation reportedly around 0.8–0.9 (according to anecdotes, many say it's very accurate). That suggests their scoring and possibly underlying item calibration is strong. Another outcome: students often credit UWorld for their passing or honors performance, but that's partly due to the content coverage and practicing of high-yield topics. There's an expectation that going through UWorld (which is not short – ~2000+ questions) can increase Step 1 scores significantly (some say by dozens of points).

Applicable ideas: One takeaway is UWorld's interface for explanations – each question comes with a mini-textbook explanation and often tables or images. Our system should similarly provide rich explanations or at least links to them (like AMBOSS or open sources) after each question, because that's when students are most receptive to learning (just after making a mistake or guess). UWorld also allows customizable quizzes (timed mode, tutor mode). We could incorporate a *mode where the adaptivity is toned down* (tutor mode showing explanations immediately) vs *exam simulation mode* (adaptive difficulty but no explanations until the end, to mimic test conditions). UWorld's lack of sequencing adaptivity implies that adding even basic adaptivity (like focusing on weaknesses or spacing practice) would be an improvement for users, as long as they still feel in control. Thus, we should allow users some control (e.g., the ability to manually select a subject if they want, or override the suggestion if they feel like doing a particular topic). User agency is important to gain trust – that's something to balance because fully algorithmic can sometimes frustrate users who have their own study strategies.

Anki (Spaced Repetition Software)

Adaptivity approach: Anki is the quintessential example of spaced repetition in practice. It's *highly adaptive on scheduling*: each flashcard's review interval is personalized based on the user's feedback ("Again, Hard, Good, Easy"). It implements the SM2 algorithm (with modifications) which essentially adjusts the interval and ease factor per card. The algorithm ensures that difficult cards are shown more frequently and easy cards less so, targeting that the next time you see the card, you'll be on the verge of forgetting (optimal point for reinforcement) [17]. Over time, each card's intervals might range from minutes to months to years, depending on how well the user remembers it. Anki does not automatically decide *what* content to learn (users create or download decks), so it's not adaptive in content selection, only in review timing. But because med students often make huge Anki decks (sometimes tens of thousands of cards covering the whole syllabus), Anki's impact on medical education is huge. It virtually guarantees you won't forget things – if you stick with the reviews.

Outcomes: As mentioned earlier, studies of Anki use in med school show significant performance benefits. One study (Wright State Univ.) found that students who used Anki and had many mature cards had **exam scores ~11 percentage points higher** on the NBME Comprehensive Basic Science Exam than those who didn't [19] [20]. Another survey study noted that spaced repetition was associated with higher exam scores and class ranks [57]. Many top-performing students attribute their success to Anki enabling long-term retention of details. However, Anki's weakness is that it's only as good as the cards you have – if a student doesn't cover a topic in their deck, Anki won't magically help. Also, over-reliance on flashcards can sometimes lead to rote memorization over deep understanding (a criticism occasionally raised). But by and large, **retention** is where Anki shines – it essentially eliminates forgetting for those who diligently review, as evidenced by personal reports of remembering even obscure facts on exams because "it was on a flashcard I reviewed last week."

Interface and strategy: Anki's simplicity is key: every day it tells you exactly which cards to review (based on the algorithm). This reduces planning overhead for the student – they just do what the schedule says. Our system can emulate this by having a "Review due" queue (driven by the per-user decay model). This ensures the student always knows what to do for spaced practice, in addition to new learning tasks. Also, Anki's feedback buttons (Again/Good/Easy) let the user calibrate the system's estimate of difficulty. We might incorporate a confidence rating from the student after a question ("Did you guess? Know it cold? etc.") to refine our model – similar to how Anki asks you essentially to rate difficulty. This could feed into the memory model (e.g., if the student felt it was "easy," increase the half-life more).

Another aspect: the open-source community around Anki has produced advanced algorithms like **FSRS (Forecasting Spaced Repetition Schedule)** which apply machine learning to optimize scheduling, reportedly improving efficiency by a further margin. Our system can benefit from these innovations by possibly integrating such algorithms or at least keeping the scheduling flexible to improvement.

One caution from Anki: review burden management. If a student has thousands of due items in a day, they get overwhelmed and might drop off (the "ease hell" scenario where too many lapses make it brutal). A good adaptive engine needs to modulate the load – maybe by capping reviews per day or dynamically adjusting the algorithm's strictness to ensure the user isn't demotivated by huge queues. Real adaptivity includes adapting to the *user's capacity and schedule*. Anki leaves it to the user to maintain or suspend decks when overloaded; our system could be smarter – if it sees the student consistently failing to complete

reviews, it might prioritize the highest-yield items or temporarily slow down new material until backlog is manageable.

NBME and Formal Exams

Approach: The National Board of Medical Examiners (NBME) is not a learning platform but rather the body that produces exams like the USMLE Step 1, subject exams, etc. However, understanding NBME's methods informs our engine in terms of **assessment accuracy and standards**. NBME uses psychometric methods (IRT) to calibrate exam questions and equate scores across different exam forms. They often pre-test questions as unscored items to collect stats. In practice, NBME's scoring means that if a student gets a very hard set of questions, the scoring accounts for that (versus an easier set). They ensure that a 70 on one form is equivalent to a 70 on another in terms of percentile ability. For our engine, aligning with NBME's calibrated difficulty can make our feedback more relevant. For example, if our engine can say "Your estimated Step 1 score is 230" using an NBME-like scale, students will find that meaningful. Achieving that requires IRT calibration of our items against a reference. One way is to include some retired NBME questions in our system and use them as anchor items (if legally allowed or via user data). Another is to allow students to take official NBME practice exams and input their results, which we then use to fine-tune our model's mapping of ability to NBME score.

NBME Practice Exams and CMS: NBME offers Comprehensive Basic Science Self-Assessments (CBSSA) and Clinical Mastery Series (CMS) for clerkships. These are fixed-form tests but scored with equated scales. Many students take these; performance on them is often considered a predictor of actual exam. These tests essentially serve as *ground truth evaluation* – they don't adapt but they provide a snapshot of ability. Our engine can integrate with these by possibly recommending when a student should take a practice exam and using the result to calibrate their profile (for instance, if our engine predicted the student was at 60th percentile but NBME practice shows 40th, we recalibrate some difficulty assumptions).

Outcomes: NBME's concern is the validity and reliability of assessment. In that vein, our system's adaptive assessments (like the adaptive quizzes using IRT) should aim for high reliability (say >0.90 reliability with 40 questions, similar to an NBME exam) and validity (correlate with actual performance). NBME has published that their subject exams and Step exams have reliabilities around 0.8-0.9 (due to many questions). We might not reach that with fewer questions, but adaptivity can compensate by targeting difficulty to reduce uncertainty. If our engine can demonstrate an adaptive test that correlates, say, 0.85 with NBME scores in a pilot, that's a big success – it means students can rely on our predictions.

Ideas from NBME: NBME uses **feedback like content area performance** (you get a report saying relative performance in each discipline/system). We should produce similar reports after enough data – e.g., "Your performance in Cardiovascular is above average, in Behavioral Sciences below average" compared to a cohort. This requires normative data or at least a large user base to compare, but even a descriptive "mastery per area" helps. Another NBME aspect is **standard setting** – determining what ability is considered passing. We might incorporate a notion of a mastery threshold per concept that corresponds to "ready for exam" level (like 85% mastery might be our threshold, analogous to NBME's passing standard).

Additionally, NBME's questions are case-based and multi-step reasoning often. That reminds us that adaptivity isn't just about difficulty and timing, but also *thinking process*. Perhaps in future, adaptive engines could detect if a student consistently falls for certain distractor types or reasoning errors and then give feedback or specific practice on that. NBME does this implicitly by analyzing item responses; we could do

something similar if we tag distractors with misconception labels. For example, if a student always picks the option that indicates confusion between two diagnoses, we identify that pattern and intervene with a teaching point. This goes beyond NBME, but is facilitated by collecting per-option data, which we should.

Open-Source and Notable Projects

Several GitHub projects and research prototypes align with our goals:

- **OATutor (Open Adaptive Tutor)**: This is an open-source intelligent tutoring system that uses BKT for skill mastery and has a content recommendation component [58]. It likely demonstrates how to integrate BKT in practice with a simple UI. Examining its code could provide patterns for structuring our BKT states and selecting next problems (though our domain is different, the software principles carry over).

- **EduAdapt AI** (mwasifanwar/eduadapt-ai on GitHub): This project, as seen in its README, outlines a very similar vision to ours [59] [60]. It mentions combining knowledge tracing, content recommendation, and reinforcement learning for personalized learning journeys. Notably, it uses **FastAPI**, LSTM-based knowledge tracing (DKT), and a reinforcement learning optimizer for learning paths [61]. The system architecture diagram in their README (reproduced in our Figure 1 textual summary) maps out layers: from student interaction to knowledge tracing to a personalization engine, then to reinforcement learning for optimal recommendations [60] [62]. This confirms that our planned architecture is on the right track. The project also mentions a **content graph for concept relationships** and an **adaptive quiz generation with difficulty scaling** [61] – exactly our plan with Neo4j and IRT. That project is early-stage (only 3 stars on GitHub) but serves as a sanity check that our multi-layer approach (KT + RL + content graph) is feasible. We can potentially reuse components or ideas from there, such as the FastAPI setup or how they define the state space for RL.

- **pyBKT** (Python BKT library): This is a tool for fitting and using BKT models [63]. We likely will not require heavy fitting since we can calibrate BKT parameters from literature initially (e.g., initial knowledge = maybe 0.3 if novices, learn rate ~0.2 per practice, slip ~0.1, guess ~0.2 – these are typical ranges [1]). But pyBKT could be integrated for periodically re-estimating those parameters from our data, or to try IRT-enhanced BKT variants (the MDPI paper [49] discusses variants of BKT that incorporate item difficulty, effectively blending IRT with BKT).

- **Algorithms from Duolingo**: Duolingo open-sourced the **HALF-LIFE REGRESSION** implementation and dataset [64] [24]. We can use that code as a basis for our spaced repetition model. Also, Duolingo's **Birdbrain** AI (not open source) was described to use an ensemble of logistic regression and Neural networks to guide difficulty and content for lessons. That suggests that in production, a mix of these techniques (not purely deep learning) is effective at scale [22].

- **MDSteps**: This is a newer commercial platform directly comparable (as it claims to be an "adaptive QBank + AI tutor"). The MDSteps blog highlights features like an *Automatic study plan*, *Adaptive QBank with 9000+ questions*, *personalized difficulty curves*, *readiness dashboard*, and generating flashcards from misses [65] [66]. They explicitly mention the engine keeps you in a "productive struggle" zone by adapting difficulty [36], and that this stabilizes confidence and yields consistent gains. They also integrate *AI tutor and analytics* to behave like a personal coach [67]. The existence of MDSteps validates our concept; their marketing even aligns with our approach (targeting weak spots, adaptive

daily plan, integrating flashcards). If MDSteps publishes outcomes, it might be in terms of user satisfaction or anecdotal improvements (they haven't been around long enough for large studies). However, their emergence suggests that the market expects such adaptivity beyond static banks like UWorld and Amboss.

From MDSteps we can glean some interface ideas: they tout a *"Depth-on-Demand™ Explanations"* which likely means the explanation can start concise but you can dig deeper if needed (maybe collapsing/expanding rationales) [68]. They also mention *"readiness signals"* on a dashboard [69] – possibly something like green/yellow/red indicators for each subject to show how exam-ready you are. We should include a similar dashboard in our UI to give students a quick sense of progress. MDSteps also automatically makes flashcards from questions you got wrong [70], presumably for later review – a neat feature to incorporate (in our system, since we already track those as items to review, we just need to present them in a flashcard mode if user wants).

In summary, competitor and project analysis reinforces that: (1) **Personalization and adaptivity are key differentiators** (AMBOSS and MDSteps promote it heavily), (2) bridging QBank with content library is valuable (AMBOSS style linking), (3) spaced repetition is recognized as vital (AMBOSS integration with Anki, MDSteps including flashcards, Anki's proven use), (4) difficulty adaptation to keep students in the optimal zone is beneficial (MDSteps, Duolingo, our ELO/IRT plan), and (5) analytic feedback loops (dashboards, predictions) help motivate and guide users (UWorld's performance stats, MDSteps readiness). We will leverage these insights in implementation: e.g., ensure our front-end has a **Progress Dashboard** indicating mastery per subject, predicted exam score, etc., and a **Daily Plan** that clearly tells the student what to do (new questions, due reviews, suggested readings – removing uncertainty from their study schedule).

Implementation Plan and Architecture

Our technology stack will comprise a **FastAPI** (Python) service for machine learning components, a **Go service** for high-performance API needs and orchestration, a **React/Next.js** front-end for the user interface, and a suite of storage solutions: **PostgreSQL** (for persistent data and relational info), **Redis** (for fast caching of user state and real-time queues), **Neo4j** (for representing the knowledge graph of concepts), and **Snowflake** (for data warehousing and offline analytics). **PyTorch/TensorFlow** will be used to build and train any ML models (like deep knowledge tracing or complex regressions) offline or in the FastAPI service.

The overall architecture will be microservice-oriented, with clear separation of concerns:

- **Web Frontend (React/Next.js)**: This is the client through which students interact. It will handle routing (e.g., Study dashboard, Quiz page, Review page, Progress reports) and will communicate with the backend via RESTful or WebSocket APIs. Next.js allows server-side rendering for fast initial loads and SEO (if needed for any public pages). The front-end will maintain minimal state – mostly UI state – with educational state coming from the backend on request (to prevent cheating or hacking with answers, etc.). Key components here: a Quiz Component (render question stem, options, accept answer, show explanation), a Dashboard Component (graphs of progress), and possibly a Flashcard Component (if we implement a separate UI for quick review cards).

- **API Layer**: This consists of FastAPI and Go services. Why both? We can assign **FastAPI (Python)** to any routes that require heavy ML computations or complex logic that we'll primarily write in Python

(which has rich ML libs). For example, an endpoint like `/next-question` would call into our Python logic that does BKT update, bandit selection, etc., then returns the next question. Also endpoints like `/submit-answer` would be handled here to apply the ML model updates. **Go** can be used for simpler high-concurrency endpoints that might involve straightforward CRUD or high throughput with minimal latency. For example, if we have an endpoint to stream a lot of past results for the dashboard, Go could fetch from Postgres quickly and stream it. Or if we decide to use WebSockets for live quizzes (maybe a tutor monitoring in real-time?), Go's concurrency might help. However, to avoid over-complication, we could start with FastAPI for most things, and gradually migrate performance-critical paths to Go as needed.

Another possibility is splitting by function: a **User Service** in Go (handling authentication, user profiles, session management) and a **Learning Service** in Python (handling all learning logic). They would communicate internally (possibly via REST or gRPC). This separation ensures that the Python ML service can be scaled or modified independently (e.g., deploying new model versions) without touching auth or other basics. The Go service can be the entrypoint for the front-end, forwarding or coordinating calls to Python service as needed.

- **Database (PostgreSQL)**: Will store core data including: Users (accounts, permissions), Questions (with fields: ID, text, options, correct answer, difficulty, etc.), Answers/Attempts (logging each attempt by a user: user_id, question_id, timestamp, correct, response time, maybe what answer chosen for analysis), and any other relational data like the mapping of questions to concepts (though that might also be in Neo4j). Postgres is reliable for such structured data and can be optimized with indexes for query patterns (like retrieving a user's last attempts, or fetching all questions of a concept with certain difficulty). We may also store the IRT parameters here (b, a, c for items) and ELO ratings (though ELO might mostly live in Redis if frequently updated). If we need to do complex analytic queries on this data, we'll offload to Snowflake, but Postgres will handle transactional updates (each answer recorded).

- **Redis**: Acts as an in-memory store for fast access to user state. We can store each user's current BKT mastery vector, current ELO ratings, and pending review queue here. This avoids recomputing or reloading from DB on each question. For instance, when a user logs in and starts a session, we load their state from Postgres (or reconstruct from history) into Redis. As they progress, we update Redis. We can also use Redis for caching question selection pools (e.g., a sorted set of questions by difficulty for a given topic, to quickly pick the nearest difficulty question). Redis can also implement a lightweight queue for spaced repetition: e.g., use Redis Sorted Sets where score=next review timestamp, member=card_id, so we can query for all due items easily. Another use: rate limiting or session tracking (like to ensure a user doesn't overload the system or multi-session conflict). Redis might also be used for any publish/subscribe if needed (like sending real-time progress updates to front-end via WebSocket).

- **Neo4j**: Will store the **knowledge graph** of medical concepts and their relationships. Each concept (e.g., "Beta-blockers pharmacology" or "Cranial Nerve Anatomy") is a node, and relationships might include "prerequisite" (concept A must be understood before B), "related to" (concept A and B are often used together), or "part-of" (hierarchical relationships like concept grouping). Questions can either be nodes or have relationships to concept nodes (we might not store questions in Neo4j but rather store in Postgres with a concept_id, and in Neo4j just keep concepts). The reason to include Neo4j is to enable advanced recommendations: for example, if the student is struggling with

concept B, the system might recommend reviewing its prerequisite A (if BKT shows A is shaky too). Or when a student finishes a block of content, we can find connected concepts to reinforce interdisciplinary links (like connecting physiology and pathology of an organ system). Neo4j queries in the backend could be: find the neighbor concepts of X that are not yet mastered, to suggest to the bandit as possible next things. Also, if we expand to include content linking (like recommending reading material), we could link concepts to resources (videos, articles). Neo4j can handle these associative recommendations beyond the linear skill model of BKT. Implementation wise, we'll have a Neo4j database populated with nodes for each topic from the curriculum outline (we can use the PMDC or any standard curriculum as a base). The Python service can query Neo4j via a library (py2neo or Neo4j Bolt driver) whenever needed. Performance is fine for moderate graph sizes (medical domain might have a few thousand concept nodes, which is trivial for Neo4j).

- **Snowflake (Data Warehouse)**: All interaction data will be periodically ETL'd (extract-transform-load) into Snowflake for deeper analysis. Snowflake will hold large fact tables like "UserAttempts" (with millions of rows over time) which would be slow in Postgres to aggregate for analysis but Snowflake is optimized for that. We'll use it to run training jobs and analytics: for instance, to train a new IRT model, we might pull the last 3 months of attempt data from Snowflake (which can handle the large scan) and run a PyTorch script to fit item parameters. Or to evaluate the efficacy of our system, we might compute in Snowflake the average learning gain per user per week, or retention rates, etc. Snowflake also ensures we keep a history of data even if we purge some from the main DB for performance. It can also integrate with visualization tools for dashboards for the team (maybe not directly for students, but internal KPIs). We'll set up nightly jobs (perhaps using something like Airflow or Snowflake tasks) to batch insert new records from Postgres to Snowflake, or we stream events to Snowflake using something like Snowpipe for near real-time.

- **Machine Learning Model Hosting**: For real-time inference, many algorithms (BKT, ELO, bandit selection) are lightweight and can run within the FastAPI process (just code and some in-memory state). If we incorporate a heavier model (say a neural network for knowledge tracing – DKT), we might load that model at FastAPI startup and use it per request (PyTorch can predict in a few milliseconds typically, and our scale likely manageable). Alternatively, if we do separate services, we could containerize a model server (like a TorchServe or TF Serving) that FastAPI calls, but that might be overkill initially. Instead, we can directly integrate with PyTorch inside FastAPI, given careful use of async to not block event loop (or run model in a threadpool as FastAPI's normal sync function would do). We should however separate training vs inference: training of models (like periodically retraining a GBM on updated data, or running batch optimization for IRT) can be done offline or in background tasks, not to slow user requests. We might have a separate process (or even done in Snowflake using SQL for simpler models) to update model parameters daily. Once updated, those parameters (e.g., new IRT b values) can be stored in Postgres and loaded by the API service. This micro-batch training approach means we don't try to do heavy learning on the fly in response to one user's action (except trivial updates like ELO delta or BKT which are formulaic).

- **Integration of algorithms to services**:

- **BKT, ELO, Memory (Spaced)**: These can be implemented as part of a Python module in the FastAPI service. BKT and memory schedules for each user are essentially state that we update and store (in Redis and occasionally persist to Postgres for backup). The logic for updating and querying these will be functions within the service. For example, `update_models(user_id, question_id,`

`correct, response_time)` function will handle: fetch user state from Redis (or compute if not present), update BKT (some simple math), update ELO (like a few lines), update scheduling (maybe adjust next review in Redis sorted set), then write the attempt to Postgres and also update any aggregate counts. This function is invoked on answer submission.

- **Bandit**: The bandit policy might be a learned model itself (maybe a contextual bandit with a learned value for each arm context pair). We could implement a simple bandit (like epsilon-greedy where we manually tune some exploration fraction) or use a library for contextual bandits. There's libraries like Vowpal Wabbit or various reinforcement learning libraries. However, to keep things controllable, we might start with a heuristic approach with bandit flavor: e.g., always do some interleaving of review vs new questions (like 80% new, 20% review, as research suggests some interleaving improves retention), and within new, alternate topics to distributed practice. This rule-based policy is simpler initially. As we gather data, we can formalize a contextual bandit: context features could be (current mastery distribution, time of day, streak of correct answers, etc.), arms as defined earlier. We could train a model (like logistic regression or small neural net) to predict reward given context & arm, updating it with new data (this is basically online supervised learning, which can approximate bandit optimization if we keep exploring). Actually implementing a full reinforcement learning algorithm (like policy gradient) might be complex for a cold start – better to simulate in a sandbox first. So practically, the bandit logic will likely live in the Python service as well. We can define it as a class with a method `choose_action(context)` that returns an action. It might rely on parameters that we update offline (like if we find certain arms generally have higher payoff, encode that). The bandit also can be configured to do simple things like "if any review is overdue (past due date by >1 day), do at least one review now" – a constraint to ensure urgent reviews aren't perpetually ignored by an explorative bandit.
- **Logistic/GBM Predictions**: We will train these models in the background (maybe using Snowflake or a separate Python training script triggered daily). Once trained, we can either: (a) incorporate the model into the FastAPI app for inference (e.g., load the model weights into memory). For logistic regression, that's trivial (just a weight vector). For GBM, we can use something like the `m2cgen` library to generate code from a trained model or use the model's Python object if using XGBoost's Python API. These predictions might be used when the user requests their dashboard: e.g., compute "predicted score" on the fly. Or if heavy, we compute predictions offline and store them so the API just fetches "we predict you'd score 72% on the final". Likely we will do predictions on-demand though, since it's a quick calculation for one user. In any case, the training (which might involve scanning through many user records) is what we use Snowflake for, and the result is either a set of model coefficients saved in Postgres or a model file accessible to the API.
- **IRT model**: Calibration of IRT parameters is done offline by analyzing large numbers of responses (could use a library like `pyirt` or do our own gradient descent or use Stan for 2PL models). This would run maybe monthly if we have enough new data, or any time we add a lot of new questions and get some data on them. Once new parameters are estimated, they are updated in the Postgres `Question` table (fields diff, disc, guess). The FastAPI service when selecting questions can use these updated values. If we implement an adaptive test mode using IRT to estimate ability, that could be done by a separate module that picks items maximizing Fisher information. This could be triggered when user starts a "diagnostic test" – that logic can live in Python too (just using the IRT data from DB to choose each next question based on current provisional θ estimate).

Data flow architecture: To tie together, here's how a *deployment* and *communication* might look: - The React app calls, say, `POST /api/answer` with {question_id, user_answer, time_taken}. This hits the Go API gateway. Go verifies the session/user, then calls an internal endpoint or function in the Python ML service (perhaps via HTTP or we could embed a Python interpreter but let's say HTTP for clarity). - The FastAPI

service has an endpoint `/internal/answer` that expects the data. It runs the `update_models` logic: update BKT/ELO (Redis), record attempt (Postgres), etc. It then returns a result which could include {correct: true/false, explanation: "...", mastery_updates: {...}}. Go receives this and forwards to frontend (or the front-end might have separate request for explanation; but anyway). - Then the front-end triggers the next question fetch, or we could bundle it: our answer response could already include the next recommended question ID (to streamline). Alternatively, the front-end after receiving answer outcome calls `GET /api/next` to get the next question. That hits Go, which calls FastAPI's `/internal/next` endpoint. - FastAPI's `/next` will gather context (from Redis: mastered topics etc.), run the bandit decision to choose topic/arm, then query Postgres/Neo4j for an appropriate question. Suppose it picks question with id=12345. It marks that as "in use" maybe (to avoid immediate repetition), and returns the question data (stem, options, any media links). - Front-end displays question 12345. User answers, cycle repeats.

Throughout, key state is in Redis (for speed) with backup in Postgres. If Redis goes down or we restart service, we can rehydrate from Postgres (though slight learning data might be lost since last persist, but losing a small amount is okay; we can persist critical state like total mastery maybe after each major change or every few minutes asynchronously).

Services and languages hosting algorithms: - **Python/FastAPI**: will host **BKT, ELO, bandit, memory scheduling, logistic/GBM inference, and any deep model**. Python is chosen for these because of the extensive libraries and ease of writing such algorithms. PyTorch for deep models (like if we later do DKT or use LSTM for time series of student performance). SciPy/NumPy for numeric things (like perhaps a custom implementation of IRT MLE ability estimation). - **Go**: will handle **concurrency and simplicity for non-ML endpoints**. This includes serving static content (if any), handling user login/registration (which involves password hashing, tokens – easily done in Go), and possibly managing websockets if we have any real-time features (Go's goroutines are good for maintaining many connections, though Python's async could too). If any heavy data transformation is needed that's pure IO (like moving data from Postgres to Snowflake), we could have Go tasks or just cron jobs – but that's more dev-ops pipeline. - **Go and Python Communication**: If the separation is done, we'll need to define how they talk. Easiest is to have FastAPI run on a different port or internal URL, and Go makes HTTP calls to it. This adds a bit of latency but if internal network, it's small (<< 10ms typically). We could also consider gRPC for more typed, efficient calls between Go and Python (there are Python gRPC servers). gRPC would allow streaming too, if needed (maybe not needed here). Simpler is just to do REST calls with JSON. Given adaptivity is sequential, a few ms overhead is acceptable (most of the response time is likely user think time seconds or network to user which is maybe 50-200ms). - **Containerization**: We will containerize these services (Docker for Python app, Docker for Go app, etc.), use Kubernetes or similar to deploy so they can scale horizontally if needed (likely not huge user base at first, but design for scaling). - **State and concept data stores**: - Postgres container or managed instance to store persistent state. - Redis maybe managed or container with persistence turned on (for emergency, though losing Redis isn't critical if we can recompute, but for memory model scheduling we might want AOF persistence). - Neo4j likely as a separate service, accessed via its Bolt port. - Snowflake is cloud SaaS, so we connect to it via its connector from a secure environment (not directly accessible from front-end). - **Security & privacy**: Very important given it's student data. We'll ensure communications are HTTPS, authentication tokens (JWTs) for API calls. Data like performance can be sensitive (e.g., a student wouldn't want their weaknesses exposed). So strict access control: only the student (and maybe authorized faculty if we integrate that) can view their data. We comply with any data protection laws (maybe local ones). All PII (like

name, email) is stored hashed/encrypted as needed. Also, since it's med domain, maybe not as sensitive as patient data, but still academically sensitive.

- **Integration with external tools**: Possibly integrate with Anki (like allowing export of flashcards or import from Anki decks), that might be later. Or integration with LMS (if this is used in a school, maybe tie into Moodle or an LMS via LTI), but that's beyond scope for now.

To summarize allocation: - Python: complex logic (knowledge tracing, scheduling, selection algorithm, model training). - Go: high-level API orchestration, possibly heavy concurrent tasks (like pushing notifications or handling hundreds of simultaneous answer submissions if class in session). - DBs: each chosen for what they are best at (relational, key-value, graph, analytical).

Data flow example (tying it together with tech specifics):

User opens dashboard page -> Next.js calls `/api/dashboard` (GET) -> Go service checks JWT, user id -> queries Postgres for summary of user performance (or calls Python service `/internal/metrics` for computed metrics). Possibly Go directly reads some aggregate table in Postgres (which we update asynchronously daily). Returns JSON: {mastery: {...}, progress: ..., reviews_due: N, etc.} -> React renders, including maybe "Begin Study" button.

User clicks "Begin Study" -> React calls `/api/next` (GET) -> Go calls Python `/internal/next` -> Python logic uses Redis state, etc., picks Q -> Python returns question data (maybe also an identifier for this session of question to link answer later) -> Go to React -> React displays question.

User submits answer -> React calls `/api/answer` with {question_id, chosen_option, duration} -> Go calls Python `/internal/answer` -> Python updates models, records attempt in Postgres (maybe via an async write for speed, but likely fine to commit synchronously) -> Python determines correctness and prepares explanation (from Postgres question table or a separate explanation table). It could also attach an updated mastery snapshot for front-end to display ("Mastery in Topic X is now Y%"). -> returns to Go, Go to React. React displays whether correct and explanation.

Simultaneously, the Python `/internal/answer` might trigger some background tasks: e.g., if the attempt was the Nth of the day, maybe send an event to Snowflake queue; or if user finished a whole topic, maybe schedule a congratulatory notification. But those are minor.

After explanation, the cycle continues with either auto-fetch next (maybe after a delay or user clicking next).

Architecture diagram: While not a physical image here, the system can be visualized as:

Frontend ⟷ Go API Gateway ⟷ Python ML Service ↔ (Postgres, Redis, Neo4j). Additionally, Python Service ↔ Snowflake (for periodic jobs). Redis and Postgres sync certain data. Neo4j connected to Python when needed for concept queries.

We will likely run these in a cloud environment (e.g., AWS or Azure). Snowflake is cloud by nature (likely AWS region). We ensure the services are in the same region for low latency.

Scalability and separation considerations: The chosen design is modular. If usage grows dramatically, we can scale horizontally: multiple FastAPI instances behind a load balancer (they can share Redis state, though have to be careful to avoid race conditions – but since each answer is mostly single-user operation, and each user's actions are sequential, not much conflict). Go instances can scale too to handle more concurrent clients. The database may need read replicas if there's heavy analytics, but writing attempt logs is an append-heavy workload which Postgres can handle thousands per second easily on a decent instance. Snowflake scaling is separate (it handles analytic queries by itself with its virtual warehouses). Neo4j might not need scaling beyond single node for our size, or we could use a causal cluster if needed.

Assignment of algorithms to components: A concise mapping: - BKT: Python (FastAPI service) – updated on each answer, stored in Redis, persisted to Postgres occasionally. - Decay Scheduling (Spaced repetition): Python – can be updated on each answer (set next due date), uses Redis sorted sets for schedule, a daily job (Python or even a Redis event) could wake up and if user hasn't logged in, maybe email them pending reviews or just accumulate until they come. - ELO: Python – updated on answer, quick math. - Bandit: Python – in selecting next content. Possibly use an ML model in Python to do it or a heuristic. Could also eventually be replaced by a trained policy network – then that network would be served by Python as well (PyTorch model). - Logistic regression: training could happen in Python offline (maybe as a script triggered by cron). Inference can be done inline in Python service (just dot product). - GBM: training using XGBoost in Python with data from Snowflake (could dump CSV or use connector). Inference – possibly load model in Python service or use ONNX if we want to unify (could convert XGBoost to ONNX and run with onnxruntime which is quite fast, even could be done in Go if needed since ONNX runtimes exist in many languages). - IRT: calibration offline (Python with some library or own code). Ability estimation could be integrated in Python service (e.g., for a diagnostic test, use Newton-Raphson or simple EAP estimation on the fly as student answers, that's doable with current answers). - DKT or other advanced models: If we integrate LSTM for knowledge tracing, it would be in Python (with PyTorch). Could run inference after each question by feeding the new sequence state – an LSTM updating hidden state is something we could do as alternative to BKT if we find it improves prediction. However, BKT is easier to interpret for now.

Implementing personalization in the tech stack: To illustrate, consider where each piece runs:

- **FastAPI route** `/internal/next_question(user_id)`:
- Code inside:
    1. state = redis.get(f"user:{user_id}:state") (contains mastery, ELO, etc.)
    2. context = compose_context(state, maybe also looking at pending reviews)
    3. action = bandit.choose(context) (bandit could be a simple function that returns e.g. "REVIEW" or "NEW:<topic>")
    4. If action == "REVIEW": concept = pick highest priority from redis sorted set of due reviews. If action is "NEW:<topic>": concept = that topic or if not specified, pick weakest concept from mastery.
    5. diff_target = maybe user ELO for concept or a fixed target success probability. q = Postgres query to get one question: `SELECT * FROM questions WHERE concept=... AND used_recently=false AND abs(difficulty - diff_target) = min(...) LIMIT 1;` (this might need a more programmatic selection: we can fetch a few candidate questions and choose one that best fits the difficulty). If no new question available (all seen), then fallback to review or next weak area.
    6. Mark question as chosen (maybe add to a short-term history in Redis to avoid immediate repeats).

7. Return question data (ID, text, options).

• That's executed within tens of milliseconds typically.

• **FastAPI route** `/internal/submit_answer(user_id, question_id, answer)`:

• Code:

1. Correct = check Postgres or cached correct answer for question.
2. result = (answer == correct).
3. state = redis.get(user state).
4. Update BKT: for each concept tag of the question (maybe primary concept), update mastery probability with formula. (This formula uses the BKT parameters we have for that concept, which might be stored in Redis as well or quickly fetched from memory).
5. Update ELO: read current user_rating_concept and question_rating, apply ELO update. Save back to Redis user rating, and optionally update a running difficulty rating for the question.
6. Update memory: If the question is designated for spaced review, and result is correct, extend interval (e.g., multiply by ease factor), if wrong, shorten interval. Then update its next due timestamp in the Redis sorted set. If it was a new question and result is correct and mastery is high, perhaps schedule it for future review with an initial interval (say 3 days). If wrong, maybe schedule sooner or consider showing a similar question next.
7. Write attempt to Postgres: (user_id, question_id, correct, timestamp, answer, response_time, etc.).
8. If needed, update some aggregate counters in Postgres (like increment user's total answered).
9. Save updated state back to Redis.
10. Compile response payload: {correct: bool, correct_answer: X, explanation: Y, mastery_change: Δp if any}.
11. If mastery updated significantly, we might also recompute the overall subject mastery and include that (e.g., "Cardiology mastery 85% → 88%"). That can be done by aggregating concept masteries for a subject.
12. Possibly trigger an async task: e.g., send event to an analytics queue with this attempt (for Snowflake or for real-time monitoring if any).
13. Return payload.

This architecture ensures each service/language does what it's best at. Python deals with the math and state; Go deals with concurrency and possibly faster I/O-bound tasks. The databases hold the truth and allow analysis.

In implementation, we'll start simple: likely keep everything in a monolithic FastAPI for initial development (since Python can handle a moderate number of users easily, especially with async and using uvicorn workers). Go can be introduced if performance or structural needs dictate. This incremental approach reduces complexity in early stage.

Finally, for completeness, ensure we have a strategy for **content management**: Adding new questions or editing existing ones. Perhaps an admin interface (maybe even in the Next.js as an admin page) that writes to Postgres. If difficulty or tags are changed, we'll need to update the related model components (like if tags changed, BKT concept mapping changes; if difficulty changed, that's okay we might just treat it as new IRT parameter and reinit ELO maybe). But these are manageable with some scripts or migration processes.

Overall, this plan leverages the chosen stack effectively: FastAPI for its easy integration of ML logic, Go for robustness and performance overhead tasks, and specialized data stores to maintain the complex state an adaptive learning system requires.

Predictive Performance and Expected Outcomes

Designing this engine is justified only if it leads to measurable improvements in learning outcomes and efficiency. We base our expectations on research and analogous systems:

Learning outcome gains: Adaptive learning systems have shown significant improvements in mastery and retention compared to one-size-fits-all approaches. A 2024 scoping review of personalized adaptive learning in higher education found that **59% of studies reported improved academic performance** with adaptive learning (and 36% reported increased engagement) [71] [72]. These improvements can manifest as higher exam scores, higher pass rates, or faster learning. For example, the Gates Foundation noted schools using adaptive learning technology saw a **26% increase in student pass rates** in certain contexts [73]. In our MBBS scenario, this could translate to more students passing their module exams on first attempt, or scoring higher distinctions.

Specifically, incorporating spaced repetition is expected to substantially boost long-term retention of medical facts. In controlled comparisons, students using spaced repetition recall far more information after delays than those who do not. Thus, we expect our users to retain content through the year, mitigating the typical "cram-forget" cycle. If we quantify: one could expect that on a retention test 1 month after learning, adaptive-spaced students might score 20–30% higher than non-spaced (e.g., 80% vs 50% retention), based on spacing effect literature [17]. In terms of exam performance, one study showed a **62% improvement in test scores** using an AI-powered adaptive system (Knewton) vs traditional practice [23]. That number is quite high, but even a more conservative estimate, say 10-20% relative score increase, is very meaningful in high-stakes exams (it could be the difference between failing and passing, or average vs honors).

Efficiency gains: Adaptivity should also reduce the time required to achieve competency. Because the engine targets weaknesses and avoids redundant practice on things the student knows, each study minute is used optimally. We predict that students can reach the same level of mastery in, say, 30% less time. This aligns with some findings that personalized AI learning improved learning efficiency by up to 30% [74]. For MBBS students who have a fixed amount of time to study, this means they could cover more material or have more time for clinical skill practice. Alternatively, they might spend the same time but achieve a higher mastery level (overlearning, which is insurance for exam surprises).

Exam performance prediction: Our engine will be collecting rich data on students, so we can attempt to predict their exam scores. Research indicates it's possible to predict final exam performance with reasonably high accuracy using learning data (some studies of MOOCs or courses achieved ~80% accuracy in predicting pass vs fail early on). For instance, Ivy Tech's AI pilot could identify failing students within first 2 weeks with an accuracy that saved many from failing [75] [76]. With our data, we expect to predict a student's likelihood of passing the modular exam or USMLE Step exam after sufficient practice. Perhaps after a student has answered 500 questions in our system, we might predict their Step 1 score within ±5 points. That's ambitious, but UWorld self-assessment is quite predictive, so combining our continuous ability estimation (IRT θ) with known correlations to Step scores, we can forecast with a certain confidence.

Engagement and motivation: Another outcome is keeping students engaged (not dropping off). Adaptivity can boost engagement – that review found 36% studies saw engagement rise [71]. Duolingo's experiment specifically saw a **12% increase in overall user activity** when they improved adaptivity (half-life model) [18]. We similarly expect that our engine's responsiveness (giving the right challenge) will maintain student interest and confidence, leading them to spend more time on the platform than they would on a static QBank. If on average a student using a static QBank might do 50 questions a week due to fatigue or boredom, with our engine they might willingly do 70 because it feels more rewarding and less frustrating. Over months, that extra practice directly converts to better preparedness.

Measuring those gains: We will validate outcomes through a combination of controlled experiments and observational studies: - We can run an **A/B test** where one group of students uses the full adaptive engine and another uses a "static" mode of our platform (e.g., where they just choose questions from a list without our recommendations or spacing). After a period, compare performance on a standardized test or retention quiz between groups. - We can also compare against historical data: if previous cohorts (without our engine) had an average exam score of X, and our cohort has Y, with appropriate statistical tests controlling for input GPA etc., we can see if Y > X significantly. - **Engagement metrics**: track daily/weekly active usage, number of questions practiced, etc. We hypothesize those will be higher for users in adaptive mode.

Predicted numbers: As a summary of expected gains: - *Exam scores*: maybe 5-10 percentile points higher on modular exams for average students, more for bottom quartile (who benefit most from guidance). - *Pass rates*: If historically 85% pass, maybe we can push to 95% pass by addressing each student's weaknesses proactively (as supported by adaptive learning seeing 17% increase in pass rates in one analysis [77]). - *Retention*: On a follow-up test 1-2 months after course, adaptive-trained students will remember perhaps 20% more than non-adaptive (based on spacing effect magnitude). - *Time savings*: Possibly a student might cover material in 2/3 the time it would normally take, or conversely, in the same time cover more material. - *User satisfaction*: We expect positive feedback as 92% of learners in one survey preferred adaptive environments over static [78] [79]. If our system is done right, students will *feel* the difference – e.g., "I love that it knows what I need to review – I don't have to figure out what to study next, it's like a personal tutor guiding me" (qualitative outcome).

We will aim to quantify these with user studies. For instance, do a memory recall test at semester end to see if those who followed the engine's plan recall more than those who studied on their own. Or track NBME practice test scores: does usage of our system correlate with higher improvement from baseline test to final test?

Validating predictions: We must ensure our claimed improvements are real and not just because the engine makes things easier. That means focusing on **real learning** indicators. We will validate with: - **External exams**: The ultimate test is performance on actual exams (university exams, licensing exams). If users of our system outperform non-users significantly, that's evidence. We might get data by collaborating with a college: one class uses the system, one doesn't, compare exam results controlling for prior GPA. - **Knowledge retention tests**: design pre and post tests, and maybe delayed post-test after some time. The adaptive group should show less drop-off in scores after a delay, demonstrating better retention. - **Transfer tests**: maybe give both groups some entirely new questions on similar topics to see if adaptive learners can apply knowledge better (to avoid just teaching to the test). - **Self-reported confidence and reduced stress**: There might be indirect outcomes like students feeling more confident going into exams because they have data from the system that they're ready (like the readiness dashboard from MDSteps). We can survey users about their confidence vs actual performance correlation.

We should also use our predictive models to validate themselves: if we predict someone is 90% likely to pass, did they? Calibration plots can show if our probabilities are well tuned.

By triangulating these measures, we can back up claims of improvement. Initially, during development, we'll likely rely on smaller pilot tests and simulations to ensure the algorithms behave logically (e.g., a simulated student with known learning parameters does indeed learn faster with our engine than without). As we gather real user data, we refine and measure actual impact.

Testing and Verification Plan

Rigorous testing and evaluation are critical to ensure the engine truly improves learning rather than giving an illusion of progress. We outline a multi-pronged testing strategy:

A/B Testing and Controlled Trials

We will conduct A/B tests (randomly assigning users or class sections to different versions of the system) to isolate the effect of adaptivity. For example: - **Experiment 1: Adaptive vs Non-Adaptive** – Group A uses the full adaptive engine. Group B uses a version where the next questions are random or a fixed sequence (or maybe sorted by topic but not personalized). Both groups spend the same study time or complete the same number of questions. Outcome measures: performance on a common test, retention after a delay, number of questions needed to reach a certain proficiency. Hypothesis: Group A will reach mastery with fewer questions and score higher on the test. We'll use t-tests or ANOVA to check for significance of differences. - **Experiment 2: Spaced Repetition vs Cramming** – Even within our system, we can toggle the spaced scheduling. Group A gets our spaced review prompts, Group B does new questions all the time (or reviews only right before exam). Then compare retention on a surprise quiz or exam performance in areas that were learned early on. We expect A to outperform B in long-term retention. This test ensures our per-user decay tuning is effective. - **Experiment 3: Difficulty Adaptation** – Group A gets questions matched to their ability (via ELO/IRT), Group B gets questions in a random difficulty order or fixed difficulty progression. Measure engagement (do B users give up more often when faced with too-hard or bored with too-easy?), and learning (maybe group B's high performers are fine but low performers struggle). We expect group A to have more consistent success rate and less frustration. This can be measured by metrics like attempt success rate and perhaps a survey of perceived difficulty appropriateness.

In these tests, we will also watch out for **placebo effects**: ensure that differences aren't due to any extra attention given. We can mitigate that by ensuring both groups get equivalent exposure to content and interactions – only difference is adaptivity logic. Perhaps give Group B some alternate but equally engaging interface (so they don't feel neglected).

Hypothesis Testing and Metrics

For each experiment, we'll define clear hypotheses and use statistical tests: - Increase in average score by X points – test with a two-sample t-test (if normal) or Mann-Whitney (if not). - Increase in pass rate – test with chi-square or Fisher's exact test on pass/fail counts [77]. - Engagement (e.g., days active per week) – test difference in means or distribution. - Retention measure – e.g., proportion correct on a delayed quiz, test difference. We'll set significance level (like $\alpha = 0.05$) and ensure adequate sample size to detect expected effects (power analysis). If initial pilots are small, we at least look for large effect sizes or trends.

We will also do **within-subject** analyses when possible: e.g., track each user's progress slope before and after introducing a feature (maybe we roll out adaptivity after a baseline period for all, as a paired comparison). This can help control individual differences.

Simulation and In-Silico Testing

Before deploying on real students, we will test the algorithms in simulation. We can create simulated students with predefined learning parameters (for example, set up a virtual student model that has certain probabilities of getting questions right that improve with practice, perhaps using known learning curve models). Then run our engine's algorithm on these simulated students in a sandbox: - Simulated Student S might have an initial ability θ and a learning rate – we simulate their responses stochastically. - We then check if the engine's models can accurately track the student's knowledge (does BKT's estimated mastery align with simulated ground truth mastery?). - We simulate a scenario: one group of simulated students studied with adaptivity (the engine chooses questions optimal for their model), another group studied randomly or sequentially. We measure final simulated knowledge. Because we know the simulation parameters, we can directly see which method yields better learning for these fake students. - This helps verify that under ideal conditions, our algorithms perform as intended (like a unit test for the learning logic). If, for instance, the bandit chooses weird sequences that actually reduce simulated performance, we catch that early and adjust the strategy. - We can also use simulation to fine-tune algorithm hyperparameters (like the K-factor in ELO, or the balance between new vs review questions) by seeing which settings maximize final knowledge in the simulated environment.

Historical Data Playback

If we have any historical student data (say from a prior static question bank usage), we can replay it through our models: - We can feed the sequence of a student's answers to our engine post-hoc and see what it *would have* recommended at each step, then compare to what the student actually did. This won't change outcomes (since the student did what they did), but we can identify cases where our engine would have intervened earlier. For example, maybe a student kept doing random questions and got many cardio wrong; our engine would have focused on cardio earlier – would that possibly have improved their eventual exam domain score? It's speculative but we could find patterns like "students who failed had these identifiable practice patterns that our engine would have caught." That builds confidence that the system can catch issues early. - Alternatively, if we have detailed logs from an old class (e.g., practice question performance), we can use that as a pseudo dataset to test our prediction models – train on part, predict outcome, see accuracy. If our logistic model can predict final exam scores from practice data similarly to published results, that validates that part. - Historical playback is also useful to ensure no regressions: as we tweak algorithms, we can run them on saved data to ensure they still correctly identify weaknesses and such in retrospect.

Guarding Against False Gains

A major concern: ensure improvements are real learning, not just inflated metrics. Some potential pitfalls: - **Easier questions leading to better scores**: If our engine only gave easier questions, students' percent correct would go up, but they wouldn't necessarily learn more – they'd just be coasting. We avoid this by keeping difficulty appropriately challenging. We measure *learning outcomes using external criteria*, not just the in-system accuracy. For example, even if our adaptivity yields an in-system accuracy of 80% (as it tries to maintain productive struggle), a non-adaptive might have 60% in-system accuracy. But if both groups then

take a common exam, we'll see who actually learned. We expect the adaptive group to do better on that common exam if learning was truly better. If we only looked at in-system performance, we could be fooled by adaptive making it easier; thus, external tests or at least cross-mixing of questions is needed. - **Overfitting to question bank**: Students might memorize our specific questions and appear to improve, but maybe the engine just repeated questions until they got them right, giving a false sense of mastery (a kind of "overtraining"). We combat this by having a large pool of questions and by emphasizing conceptual mastery. Also, our BKT model's guess/slip parameters help – if a student gets something right by luck (guess), BKT doesn't immediately say mastered because the guess probability accounts for that. Our IRT difficulty also ensures if they only can answer easy versions of a question but not harder ones, their ability estimate won't spike falsely. - **Placebo/UI factors**: Perhaps the adaptive group does better simply because the interface kept them more engaged (a kind of Hawthorne effect). While engagement is indeed part of the value, we want to ensure the learning is from adaptivity, not just from spending a bit more time because it's novel. We can control for time-on-task in analyses. If adaptive group spent 10% more time, and outperformed, is it time or method? Possibly both. We could do a variant where the control group is asked to spend equal time – though that's hard to enforce, we could measure and then do an ANCOVA controlling for time spent. - **User behavior**: Some might try to game the system (e.g., intentionally answer something wrong to get easier content because they're chasing some gamification metric). We need to detect and discourage such behavior. For instance, if someone keeps marking things wrong to get more fundamental reviews – maybe the solution is fine because they will just see more easy stuff and eventually have to do hard stuff to progress. Or if someone always marks "easy" on flashcards to skip them – the algorithm will naturally adjust if they start getting them wrong because the interval overshot. But we should monitor usage patterns for anomalies. A solution could be incorporate *confidence-weighted scoring*: encourage students to be honest by giving more credit for correctly indicating they know something vs guessing. Some advanced frameworks (like Bayesian testing of whether a student really knows a concept) can help ensure they truly mastered it by asking variant questions. - **Retention vs performance trade-off**: We need to ensure we measure knowledge in the long run, not just immediate performance. An engine might boost short-term scores by focusing on exam-like questions at the expense of long-term retention or deeper understanding. Our testing plan includes delayed tests to ensure retention. Also, measuring performance in application (like how well can they answer clinical vignette vs just recall facts) would reveal if we only taught shallow recall. To counterbalance, our question pool should include higher-order questions, and our adaptivity can incorporate not just factual recall but application skills (if we tag questions by bloom's level or similar, we can ensure the student also practices higher-order thinking, not only easy recall just to bump mastery). - **Real vs perceived improvement**: We will also gather qualitative feedback – do students *feel* more confident and prepared? Sometimes metrics might improve but if students feel lost without the system (dependency), that could be an issue. Ideally, the system trains them to also become self-directed learners (by modeling good review habits).

Continuous Monitoring

Once deployed, we will continuously monitor key indicators for any "false gains": - If we see an unusually high overall accuracy in the system without corresponding high external exam scores, that's a red flag (maybe the content was too easy or leaked). - If a certain algorithm update suddenly makes things look great internally, we'll double-check with an external metric. For instance, if we tweak ELO and next day everyone's mastery jumped 15%, it might be a calibration change, not actual learning – we should recalibrate rather than celebrate. - We will implement dashboards (for internal team) to track metrics like average mastery vs time, average retention quiz score, etc., to catch anomalies. For example, if memory model scheduling was too lenient, we might see retention quiz scores drop – then we tighten it. - We'll also

keep an eye on user feedback: if adaptivity is pushing too hard or too easy, students might complain it's boring or overwhelming – that qualitative data helps adjust the difficulty targeting.

Ensuring Real Learning Improvements

Ultimately, the gold standard is performance on real exams and the ability of students to recall and apply knowledge in clinical scenarios. Our verification plan includes working with educators to see if our system's focus aligns with curriculum objectives. For example, if our engine identifies someone as mastered in a topic, does the professor's assessment of them in class projects or viva align? If not, we refine what mastery means for that topic (maybe include more application questions).

We also plan to run **longitudinal studies**: track students who consistently use the system across multiple modules or years vs those who don't, and see differences in cumulative outcomes (like final exams, or ease in later courses that build on earlier ones). If adaptivity yields durable gains, those students should perform better downstream as well, not just in the immediate next test.

By designing experiments and monitoring in these ways, we aim to verify that our engine delivers *authentic* learning improvements – better understanding, recall, and application – and not just inflated practice stats. Any sign of discrepancy will prompt an investigation and algorithmic adjustment (this is an ongoing research-development cycle typical for learning platforms).

Additional Promising Algorithms and Strategies

In addition to the core algorithms we've implemented, there are emerging and hybrid approaches that could further enhance our learning engine:

- **Deep Knowledge Tracing (DKT)**: This is a neural network (typically an LSTM or RNN) approach to knowledge tracing that can potentially capture complex patterns in student learning sequences better than BKT. DKT models a sequence of student responses and learns latent representations of knowledge state without requiring predefined skills for each question. It has shown improved prediction accuracy over BKT in some studies [80] . We could experiment with DKT by feeding in our student response histories to an LSTM that outputs probability of next answer correct. This might capture interactions between concepts or forgetting patterns implicitly. For example, DKT might learn that after 5 days without practice, certain topics drop in performance significantly (something BKT doesn't model). We have to be cautious: DKT is less interpretable, but hybrid models exist (like DKT+ forgetting mechanisms). One idea is to use DKT to complement BKT: e.g., use DKT's predictions as a feature or to adjust BKT parameters over time. Or run DKT offline to discover if there are concept relationships we missed (some researchers try to extract learned embeddings to see topic groupings). Since we have PyTorch in our stack, integrating a DKT model (and even doing online learning updates to it as new data comes) is feasible. If it outperforms BKT in predicting errors, we can use it to drive content recommendation more accurately (it might tell the bandit which question the student is likely to get wrong – which could be exactly where to target for learning).

- **Multivariate or Hierarchical Models**: Instead of modeling each concept separately (like BKT does), we can consider models that account for relationships between concepts. For instance, a **Hierarchical Bayesian Knowledge Tracing** could model general ability and specific skill mastery simultaneously. Or an extension of IRT known as **Diagnostic models** (like DINA/DINO) which allow

multiple skills per question and can model the AND/OR requirements of skills. Such cognitive diagnostic models can give more detailed insight into misconceptions. If we find certain questions require multiple concepts (e.g., to solve a clinical case you need path + pharm knowledge), a multi-skill BKT or IRT might be needed. Adopting a **matrix factorization** approach (treating student and question as factors to predict correctness, akin to recommendation systems) is another way to capture multi-concept influences without tagging every skill. This could be an avenue if our tagging is incomplete or if skills overlap.

- **Reinforcement Learning (Policy Gradients / Q-learning)**: Our multi-armed bandit is a simple RL scenario (contextual bandit). For more complex planning (like deciding a whole sequence of content adaptively), deeper RL algorithms could be used. For example, a **policy gradient method** could treat the problem of maximizing the student's final exam score as the reward and try to directly learn a policy of what to show when. This would require either a simulator or a lot of data to train safely. There are some research attempts where they simulate student learning as an MDP and apply deep Q-learning to find optimal teaching policies. Given the risk of trying an untrained RL policy on real students (it might do odd exploration), we'd probably experiment in simulations first. But this could discover non-intuitive strategies (maybe it finds that after 3 easy questions, throwing a very hard one yields better long-term retention due to desirable difficulty). A safe way to introduce this is with **offline RL**: use our collected data to evaluate policies and maybe train a policy network offline, then test it in an A/B pilot. If it shows improvement, then integrate.

- **Graph-based Knowledge Models**: We already use Neo4j concept graph for prerequisite relationships. We can further leverage graph algorithms – e.g., **Knowledge Space Theory** and **learning spaces**: these attempt to model the set of knowledge states a student can have (as nodes in a graph, with edges if one state can be reached by learning one more concept). This could help in ensuring curriculum coherence – the system can detect if a student's pattern of mastered concepts is inconsistent (maybe they learned some advanced concept without a basic one, indicating perhaps guesswork or superficial learning). By identifying such anomalies, the system can fill the gaps (like "you solved some tough cardiac physiology questions but missed a basic one, let's review the basics to solidify your foundation"). There are algorithms like the **ALEKS system** in math that use model of knowledge spaces to decide next problem. We might not implement full formal learning space theory, but we can approximate by our prerequisite graph: ensure prerequisites are mastered before advanced ones, otherwise recommend going back.

- **Natural Language Processing (NLP) for content**: We could use NLP to analyze question text and student free responses (if any) to provide more adaptive feedback. For instance, if we ever let students type an explanation or reasoning, NLP (maybe an LLM or classifier) could assess their reasoning and adapt feedback accordingly (similar to how some AI tutors work with open responses). Also, NLP could help generate hints or simpler explanations on the fly if a student is struggling (though our current Qs are MCQ, hints could still help). Additionally, semantic similarity via NLP could identify related questions to recommend (like "you struggled with this question, here are 2 similar ones for more practice"). This can enrich the adaptivity beyond our explicit concept tagging.

- **Interleaving and Blocked practice strategies**: In cognitive science, how you sequence topics (interleaved vs blocked) affects learning. Our bandit might discover some pattern, but we could explicitly incorporate such strategies. For example, research suggests interleaving examples from

different categories can improve discrimination learning. We might implement a mode where questions from different subjects are mixed even if the student is focusing on one module, to improve integration (this could be especially relevant in integrated curricula like modular systems). Or use **spacing not just in time but in content** – e.g., revisit a topic after doing a different one in between. These strategies could be pre-programmed or learned. We should test these as well (maybe sometimes blocking is better for initial learning, and interleaving for retention – find the right mix).

- **Student Meta-cognitive Modeling**: We could track not just knowledge, but behaviors like procrastination or confidence. For example, a **forgetting curve model** combined with a **time-decay of engagement** could schedule content to keep motivation. Or incorporate **gamification** elements (badges, streaks) carefully as Duolingo does, to encourage consistent use (though Duolingo's approach shows gamification must not overshadow learning [81] [82]). If we detect that a student hasn't reviewed a certain subject in a long time (lack of self-regulation), the system could proactively prompt them or send a reminder, acting as a meta-cognitive scaffold. Another idea is **self-reflection prompts**: occasionally ask the student to predict their performance or rate their understanding, then show actual results; this has been shown to calibrate students' self-awareness which is important for independent learning.

- **Collaborative filtering / Peer learning**: With enough users, we might find patterns across students. For instance, if a cluster of students all struggle with a particular concept, our system could alert instructors or provide an extra tutorial on that concept to everyone. Or use collaborative filtering: "students who got this question wrong found the following explanation or analogy helpful" (if we allow sharing of mnemonics or reasoning). Not exactly an algorithm, but a feature that leverages group data. On algorithmic side, we might train models on a population to predict which content works best for which type of student (as perhaps gleaned from their performance and maybe background). That goes into adaptive content recommendation beyond questions, e.g., recommending specific videos or external resources when the system's questions reveal a gap.

- **Open Learner Models**: A strategy is to make the model of the student visible to them (in a user-friendly way), to engage them in the adaptive process. For example, showing a skill meter or a learning path can motivate them and also let them reflect "I see my surgery section is weak, I'll focus there." This is not an algorithm per se, but tying the output of our algorithms into the UI for student reflection can boost meta-cognition, which research shows improves learning (students take control when they understand their progress data).

- **Hybrid of Mastery and Mastery Speed**: Some algorithms consider not just whether a student masters something, but how fluently. In medicine, sometimes quick recall is needed (e.g., in a rapid response scenario, you must recall a drug dose quickly). We might incorporate timing into mastery – e.g., track if a student eventually gets all cardiology questions right but takes very long on them, perhaps more practice or time-pressure practice would be beneficial. This leads to ideas like **response time modeling** (some use cases: an item response model that includes latency). If we see a student knows something but very slowly, we can adapt by giving timed quizzes to build speed. That's an advanced nuance that could differentiate a basic mastery from true fluency.

- **Error Analyses and Knowledge Tracing**: We can categorize the types of errors (conceptual error vs calculation vs misreading question, etc., if detectable). If we find patterns (like a student often

narrows to two options and picks the wrong one due to a particular misunderstanding), we can adapt by giving a mini-tutorial on that distinction. This could be aided by a simple expert system or by tagging questions with common misconceptions.

- **Personalized decay factors (already done) but possibly using a model like** Leitner system with dynamic programming**: There's research on optimizing review schedules by formulating it as an optimization problem – maximize recall probability at exam time given limited time. Algorithms like** GREEDY schedule** or using search to allocate study time among items could be considered, though our bandit is a stepwise approach to that. Nonetheless, formal scheduling optimization might slightly outperform simple heuristics by solving a knapSack-like problem where each day's study minutes are allocated to items that yield the biggest increase in expected final score. That becomes complex to solve exactly, but approximate solutions or reinforcement learning can tackle it.

- **Data-driven content creation**: Over time, our analytics might find certain topics are systematically hard or certain question formats lead to better retention. We could then create new questions focusing on those gaps or use AI to generate variant questions for extra practice. For example, if many students keep failing a specific step in acid-base analysis, generating a few more questions on that specific step (with different phrasing or numbers) could help – an NLP model could help generate or an educator uses our data to write targeted content. So adaptivity can extend to content generation or recommendation of external content (like if our system sees a student struggle with ECG interpretation, it might suggest "Watch this 5-minute ECG basics video" linking to a known good resource).

In summary, after deploying the core system, we have a roadmap of enhancements: - Incorporate deeper learning models (DKT) if needed for accuracy. - Use reinforcement learning more fully to optimize long-term outcomes through simulation-driven policy improvements. - Exploit relationships among concepts via graph-based and multi-skill models. - Further personalize not just *what* and *when*, but *how* content is presented (maybe adapt presentation style to learning styles – although learning styles theory is controversial, but some adaptivity like more visual vs more textual explanation based on user preference could be tried). - Introduce social or competitive elements carefully (maybe a leaderboard of mastery points to motivate, though must ensure competition doesn't encourage gaming or demoralize weaker students – maybe only after a basic threshold achieved). - Embrace any new research: e.g., just recently they mention FSRS in Duolingo context [83] – we should keep an eye on such developments and incorporate if proven.

The beauty of our architecture is that it's flexible to plugging in these new algorithms. We designed it so that user interaction data is collected and can train new models. If a new algorithm (say a better memory model or a more accurate knowledge tracing model) comes out, we can test it on our data, and if it performs better, swap it in as the engine while keeping the interface similar. This adaptability ensures the system remains cutting-edge and continues to improve as the science of learning advances.

Conclusion: By integrating this array of algorithms – each validated by research – into a cohesive architecture, we aim to create a powerful personalized learning engine for MBBS students. The system will continuously adapt to each student, optimizing review schedules, difficulty, and content focus, much like a dedicated tutor for every learner. Drawing on lessons from existing platforms and careful experimentation, we expect significant gains in learning efficiency, retention, and ultimately exam performance. This design

not only addresses the immediate needs of exam preparation but also fosters habits of spaced practice and self-monitoring that benefit lifelong learning in medicine. We have laid out how to implement, test, and refine this system, and with ongoing research integration, it can remain at the forefront of educational technology innovation.

Sources:

1. Bulut, O., et al. (2023). *An Introduction to Bayesian Knowledge Tracing with pyBKT* – **explains BKT foundations and variants** [1] [50].
2. Pelánek, R. (2016). *Applications of the Elo rating system in adaptive educational systems* – **shows ELO is simple, robust, and comparable to IRT/BKT** [29] [27].
3. DABSEC (2023). *Sequencing Educational Content Using Diversity Aware Bandits* – **bandit algorithms effective for educational sequencing** [37] [38].
4. Duolingo Team (2016). *How we learn how you learn* (Duolingo Blog) – **half-life regression model improved retention (daily activity +12%)** [18] [22].
5. Axon Park Blog (2023). *How Effective is AI in Education?* – **Knewton adaptive learning users improved test scores by 62%** [23].
6. duPlessis et al. (2024). *Personalized adaptive learning in higher education: A scoping review* – **59% of studies showed improved performance with adaptive learning** [71] [72].
7. Kuepper-Tetzel, C. (2014). *Spaced repetition and learning* – **spacing effect robust for long-term retention** (conceptual support mirrored by Duolingo and Anki studies).
8. Anki usage study (2023). *Exploring the Impact of Spaced Repetition Through Anki on Exam Performance* – **med students using Anki scored ~11.5 percentage points higher on exams** [19] [20].
9. MDSteps Editorial (2025). *MDSteps vs UWorld vs AMBOSS* – **MDSteps emphasizes adaptive QBank keeping students in productive struggle zone, improving confidence and consistency** [31] [32].
10. Adaptemy (2023). *Metrics that matter in adaptive learning* – **learner preference for adaptive (92%) and engagement up 20%** [78] [74].
11. Slater & Baker (2018). *BKT model evaluation* – **typical BKT AUC ~0.74 for next problem correctness** [6].

---

[1] [2] [3] [4] [8] [9] [10] [49] [50] An Introduction to Bayesian Knowledge Tracing with pyBKT | MDPI
https://www.mdpi.com/2624-8611/5/3/50

[5] Tracing Students' Learning Performance on Multiple Skills using ...
https://dl.acm.org/doi/10.1145/3551708.3556202

[6] [7] Bayesian Knowledge Tracing (diagram reproduced from Author, 2015) | Download Scientific Diagram
https://www.researchgate.net/figure/Bayesian-Knowledge-Tracing-diagram-reproduced-from-Author-2015_fig1_329249607

[11] [12] [13] [14] [15] [16] [17] [18] [22] [24] Duolingo Blog
https://blog.duolingo.com/how-we-learn-how-you-learn/

[19] [20] [46] [47] Exploring the Impact of Spaced Repetition Through Anki Usage on Preclinical Exam Performance - PMC
https://pmc.ncbi.nlm.nih.gov/articles/PMC12357012/

[21] [57] Spaced repetition and other key factors influencing medical school ...
https://link.springer.com/article/10.1186/s12909-025-07605-w

23 75 76 How Effective is AI in Education? 10 Case Studies and Examples - Axon Park
https://axonpark.com/how-effective-is-ai-in-education-10-case-studies-and-examples/

25 openlifescienceai/medmcqa · Datasets at Hugging Face
https://huggingface.co/datasets/openlifescienceai/medmcqa

26 27 28 29 30 33 34 35 51 52 53 54 Applications of the Elo rating system in adaptive educational systems - ScienceDirect
https://www.sciencedirect.com/science/article/abs/pii/S036013151630080X

31 32 36 65 66 67 68 69 70 UWorld vs AMBOSS vs MDSteps: Which USMLE Prep is right for you? — MDSteps
https://mdsteps.com/articles/general-exam-prep/uworld-vs-amboss-vs-mdsteps-which-usmle-prep-is-right-for-you

37 38 39 40 41 42 Sequencing Educational Content Using Diversity Aware Bandits
https://educationaldatamining.org/EDM2023/proceedings/2023.EDM-posters.57/index.html

43 44 45 arxiv.org
https://arxiv.org/pdf/2102.04250

48 Riiid Answer Correctness Prediction | Kaggle
https://www.kaggle.com/competitions/riiid-test-answer-prediction/discussion/201223

55 Empowering your Teaching. Shaping their Future. - AMBOSS
https://www.amboss.com/us/pa/programs

56 Does UWorld have an algorithm to show weaknesses? - Reddit
https://www.reddit.com/r/barexam/comments/ywcg7y/does_uworld_have_an_algorithm_to_show_weaknesses/

58 CAHLR/OATutor: Open Source Intelligent Tutoring System w - GitHub
https://github.com/CAHLR/OATutor

59 60 61 62 GitHub - mwasifanwar/eduadapt-ai: AI-driven adaptive learning system that personalizes educational content based on student performance, learning style, and engagement metrics. Uses reinforcement learning for optimal learning path recommendation.
https://github.com/mwasifanwar/eduadapt-ai

63 Bayesian Knowledge Tracing - Cognitive Learning Engine
https://acme-2d4beb34.mintlify.app/guides/bkt

64 duolingo/halflife-regression - GitHub
https://github.com/duolingo/halflife-regression

71 72 Personalized adaptive learning in higher education: A scoping review of key characteristics and impact on academic performance and engagement - PMC
https://pmc.ncbi.nlm.nih.gov/articles/PMC11544060/

73 The impact of artificial intelligence and adaptive learning ... - Medium
https://medium.com/@the_exploring_minds/the-impact-of-artificial-intelligence-and-adaptive-learning-on-the-quality-and-accessibility-of-fd09373e71a2

74 20 Statistics on AI in Education to Guide Your Learning ... - Engageli
https://www.engageli.com/blog/ai-in-education-statistics

77 The Impact of Learner Data on Adaptive AI-Driven Learning Systems
https://www.learningguild.com/articles/the-impact-of-learner-data-on-adaptive-ai-driven-learning-systems

78 79 How to Evaluate Adaptive Learning Systems: The Metrics That Matter
https://www.adaptemy.com/how-to-evaluate-adaptive-learning-systems-the-metrics-that-matter/

80 Deep knowledge tracing and cognitive load estimation for ... - Nature
https://www.nature.com/articles/s41598-025-10497-x.pdf

81 82 Duolingo’s Scaling Journey: Education Revolution or Engagement Obsession?
https://gadallon.substack.com/p/duolingos-scaling-journey-education?utm_source=substack&utm_medium=email&utm_content=share&action=share

83 Duolingo's search trend vs. user base - LinkedIn
https://www.linkedin.com/posts/dhawals_while-reading-duolingos-2024-10-k-i-came-activity-7302455251779866624-bx4z