**End-to-End Personalized Recommendation Delivery Flow**

Using the provided architectural diagrams, this report walks through the end-to-end flow of delivering personalized merchant recommendations to a Card Member. The architecture is organized into five logical layers – Card Member/Channel, API, CSRT (real-time scoring service), Batch (offline pipeline), and Cornerstone (data foundation) – which we will trace in sequence. The narrative follows a typical scenario where a user interacts with a channel (e.g. Amex Maps or a personalized email), triggering a cascade of backend processes that result in a set of ranked recommendations being returned.

*Figure: High-level architecture for personalized recommendation delivery, showing the Channel (A) and API layers, the CSRT microservice with scoring algorithms (B, C, D, E steps), and the supporting Batch and Cornerstone data layers (numbered 1–4)【6†】. The flow begins at the Channel (left) and ends with results back at the Channel, powered by data and models prepared in the Batch/Cornerstone layers.*

**Card Member/Channel Layer (User Interaction)**

This is the entry point where the Card Member (user) interacts with an American Express channel that offers personalized recommendations. Key points at this layer include:

**Channels:** Examples include the Amex Maps web feature, Dynamic Email content, MyCard Dashboard (offers section in the online account), or the Amex Mobile app. Each channel may have a specific context:

* **Amex Maps:** The user might search for nearby merchants (e.g. restaurants, shops). They could be logged in (allowing personalization) or not (in which case only generic recommendations are possible).
* **Dynamic Email:** The user receives an email with embedded personalized merchant suggestions (e.g. "Merchants You Might Like"). This is typically a logged-in scenario (the email is targeted to a specific customer, and content is personalized at send-time or open-time via an API call).
* **MyCard Dashboard (Offers section):** When a logged-in Card Member visits their account dashboard, a module may show recommended merchants or offers.
* **Shop Small or Marketing Campaigns:** Some channels correspond to specific campaigns (e.g. a "Shop Small" program page that recommends local small businesses).

**User Action:** The flow often starts with a user action or request on the channel:

* In an interactive channel (Maps, Mobile app), the user might provide input such as a location or search term (e.g. searching "coffee near me"). If the user just opens the map without input, the system may still attempt to find interesting nearby merchants.
* In a passive channel (Email), the "action" is the system deciding to present recommendations in the email, and the request to fetch personalized content is made in the background.

**Channel Request to API:** The channel then makes a request to the backend Recommendation API. This request typically includes:

* The Card Member's identifier or token (if the user is logged in) – for example, a secure customer ID or GUID associated with their account.
* Contextual parameters: e.g. the user's location (GPS coordinates or a postal code if using Maps), any search filters or keywords, the channel identifier (to let the backend know which channel is calling, since business rules may differ by channel).
* An indication of whether the session is authenticated/secured (user logged-in) or not. This is critical because it determines if personalization (using customer data) is allowed. If not logged in, the system will default to non-personalized recommendations (e.g. popular merchants) for privacy reasons.

**Example Start:** A user opens Amex Maps and searches for "dining". The Maps front-end gathers the user's location (say, downtown zip code 10001) and their Card Member ID (since they are logged in), then calls the recommendation API to get a list of dining recommendations in that area tailored for the user.

**API Layer (Orchestration and Routing)**

The API layer receives the channel's request and orchestrates the next steps, acting as a bridge between the user-facing channel and the internal recommendation engine logic. Key functions of the API layer include:

**Endpoint Reception:** The request hits an API endpoint (for example, GET /v1/recommendations or a specific "MerchantRec API"). The API layer is implemented as a microservice (or set of microservices) that validate the request and parse parameters.

**Authentication & Session Check:** The API checks if the request is from an authenticated session:

* If the user is logged in (authenticated token is present/valid), the API knows it can use customer-specific data. It will include the Card Member's ID in the downstream request for personalized scoring.
* If the user is not logged in, the API may set a flag indicating an anonymous session. In this case, certain personalization steps will be skipped. The API might route the request to a simpler logic path (for example, using a popularity-based recommendation without any customer-specific features).

**Channel Context & Routing Decisions:** The API determines the strategy based on channel and possibly experiment configuration:

* It might call a configuration service (for instance, an ECM – Enterprise Config Manager or similar) to fetch the current "control cell" configuration for this channel or user segment【4†】. This config would tell the system whether the user is part of any A/B test or control group. For example, the config may specify that 10% of traffic sees a "random" recommendation as a control (Cell C0), or that a new algorithm is being tested for a certain segment (Cells T1–T9 for different treatments).
* The API then decides which backend service or algorithm to invoke. In many architectures, this could simply be a parameter passed along to the next layer, but conceptually:
  + If the user is anonymous or in a "popularity-only" control cell, the API could route the request to a Popularity Recommendation path.
  + If the user is logged in and meant to get personalized results, the API will route to the main CSRT (scoring) service with an indication of which model/algorithm to use (e.g. "use CatBoost model").
* There may be separate endpoints or methods internally (e.g., GET /recommendations/popular vs GET /recommendations/personalized), but often a single service endpoint handles both, with logic to branch inside.

**Example:** The API sees the request from "Maps" channel with a logged-in user. According to config, 80% of such users get the ML model (CatBoost) and 20% might get a popularity-based list as a baseline. Our user falls into the 80%, so the API prepares to call the personalized scoring service.

**Request Enrichment:** Before forwarding, the API may enrich the request with any additional data available:

* It could add geo-information. For instance, if only a postal code was provided by the app, the API might call a lightweight geo-service or lookup table to translate that into latitude/longitude (unless this will be done in the next layer). This ensures the downstream service can do geo searches.
* It sets up the query parameters for the recommendation service: e.g. category filters (if the user searched "dining", include category="dining" in the criteria), location coordinates or region, the number of results needed, and the Card Member ID or a token.

**Calling CSRT Service:** Finally, the API makes a call to the CSRT (Customer Selection & Recommendation Tool) service. This is typically a REST call over the network to the microservice responsible for scoring. The payload includes all relevant info (user ID, location, channel, any search keywords, and an indicator of which algorithm or model to use, if applicable). The API layer then waits for the CSRT response.

**CSRT Layer (Real-Time Scoring Service)**

The CSRT layer is the real-time brain of the system – a microservice (or set of microservices) that performs the core personalization logic. It handles data gathering for the individual request, chooses the right model/algorithm, scores the candidate merchants, and returns the ranked recommendations. Let's break down its responsibilities step by step:

**1. Retrieve Card Member Information:**

Upon receiving the request, the CSRT service first ensures it has the necessary Card Member data (if the user is logged in):

* It may call an internal profile service or database to get the customer's basic info and preferences. In some architectures, this is done by reading from a fast data store or cache. In our case, the system leverages precomputed features from the Batch layer, so the CSRT service will fetch the user's feature vector.
* **Secure Data Handling:** The Card Member ID might be encrypted or tokenized for security. The service will detokenize it (using Voltage encryption libraries, for example) to use it internally【6†】. This allows matching to data in the feature store. All sensitive data remains protected in transit and logs – detokenization is only done in-memory for the scoring process.

**2. Control Cell & Model Selection (Routing Logic):**

Next, the CSRT applies the control cell configuration to determine which scoring approach to use【6†】:

* The service has a Router component that checks flags or config passed from the API (or looks up an experiment config by itself). It identifies if the request is in a special experiment or control group (e.g., C0 for control, T1...Tn for various test treatments).
* Based on this, it selects one of the scoring engines:
  + **CatBoost ML Model (Personalized Model)** – This is the primary machine learning model (code-named Janus for US/International versions) trained to predict a Card Member's affinity to merchants. It's typically used for the majority of logged-in personalized requests【4†】.
  + **Collaborative Filtering** – An algorithm using collective behavior data (e.g., "customers like you also liked these merchants"). This might be used for certain segments or as a complement to the main model.
  + **B2B Algorithm** – A specialized recommendation logic for Business Card Members. For example, it might prioritize merchants relevant to business spending (office supplies, travel) if the user has a corporate card. This could be a separate model or rule-based algorithm tuned on small-business data.
  + **Popularity-based** – A simpler engine that ranks merchants by overall popularity (transaction counts, trending status) rather than personal fit. This is often used for anonymous users or as a control variant. It might also be a fallback if personal data is unavailable.
  + **Random** – In some control scenarios, a random selection is returned (from the candidate set) to act as a baseline. This is typically used only in experimentation (to compare against the model's lift).

These options correspond to the Scoring methods box in the diagram【6†】. The router ensures the appropriate one is invoked. (In our example scenario, it will choose CatBoost since the user is in the personalized group.)

**3. Apply Business Rules:**

Before scoring, the CSRT service enforces any business or eligibility rules:

* For instance, it might ensure that merchants returned have no active issues (e.g., not offline or out of business – using the "active merchant" list from Cornerstone data).
* It could filter out certain categories or merchants based on regulatory or user preferences (for example, exclude gambling or adult-oriented merchants from recommendations).
* If the channel or campaign has specific rules (e.g., the "Shop Small" channel should only include small independent businesses), those filters are applied here.
* **Frequency capping / diversity:** The service might check if a particular merchant was shown to this user recently (using impression logs) and decide to avoid repeats for a while, to keep recommendations fresh.

**4. Candidate Merchant Retrieval:**

The next major step is to gather a set of candidate merchants that could be recommended, which will then be scored and ranked:

* **Geo and Contextual Query:** If the request includes location or search criteria, the service performs a lookup to retrieve matching merchants. This is often done via an ElasticSearch query on a merchant index【2†】. For example:
  + It may convert a provided postal code into a latitude/longitude coordinate (using a reference table) to do a geo-spatial search【2†】.
  + It queries the merchant index (in Elastic) with filters for the given area and any keyword/category. The Elastic index contains merchant documents with fields like location (lat/long), categories, names, and possibly tags (like "small\_business=Y/N"). This allows rapid searching of, say, "restaurants within 5 miles of 10001" or "bookstores in San Francisco".
  + The result might return a list of merchants (merchant IDs and basic attributes) that satisfy the criteria. For example, 100 merchants in the vicinity of the user.
* **Additional Filtering:** The service can further trim or adjust this candidate list:
  + It might enforce a limit (e.g., take the top 50 by some preliminary relevance or popularity score embedded in the index).
  + It ensures all candidates meet baseline criteria (must be in the "active merchant" set, etc.). If any merchant in the Elastic result is missing required data or is known to be low-quality, it's dropped.
  + If the user provided a specific filter (like "dining"), the query already scoped to that, but if it's a general request, the candidates could span multiple categories.

**5. Feature Gathering for Scoring:**

Now, for each candidate merchant, the service needs to assemble a feature vector describing the Card Member + Merchant pair:

* **Card Member Features:** The service retrieves the precomputed features for the user from the Batch layer output:
  + These features could be stored in a fast-access data store. In our architecture, the features are likely stored on a distributed file system (MapR) and accessible via a key (Card Member ID). Because MapR supports an NFS mount, the service might directly read a small file or lookup record for the user's feature profile【2†】. Alternatively, these could be loaded in-memory at startup or cached.
  + Features include things like: demographic segment, tenure, average monthly spend, top 3 merchant categories they spend in, whether they are a travel enthusiast, how often they use Amex offers, risk score tier, etc. Also online behavior like number of clicks on previous recommendations or map searches.
  + **Example:** Our user's features might reveal they spend a lot on dining and travel, have a high engagement score with the mobile app, and are in a "Foodie" segment of customers.
* **Merchant Features:** For each candidate merchant ID, the service looks up that merchant's feature data:
  + The Batch layer will have produced a profile for each merchant (or merchant location) as well, typically keyed by a merchant identifier. This could be stored in a lookup table or file on MapR, or even embedded partly in the Elastic index. If not already retrieved in the Elastic query, the service fetches it now.
  + Features include: merchant category (or multiple categories), average transaction volume and frequency (popularity among all Amex users), whether it's a "small business" or large franchise, the merchant's cluster (grouping similar merchants together, e.g. "Italian Restaurants" cluster within Dining), recency of activity (e.g., transactions in last week), and any prior engagement metrics (how often it was shown or clicked in earlier recommendations).
  + **Example:** A merchant candidate "Luigi's Trattoria" might have features indicating it's an Italian Restaurant (category = Dining/Fine Dining), it's in the top 10% popularity in that zip code for dining, it belongs to a cluster of "Italian eateries", and it participated in the Shop Small program.
* **Combined Features:** Some features may be interactions of user and merchant (though many interaction features can be precomputed offline as well):
  + E.g., "Has the user visited this specific merchant before?" (could be a binary feature, derived from transaction history – this might be computed on the fly or provided if the offline data included a per user->merchant matrix of recent visits).
  + "How far is the merchant from the user's home location?" (distance can be computed from coordinates dynamically).
  + If using collaborative filtering latent factors: the user and merchant might each have an embedding vector (from matrix factorization). The service could compute a similarity (cosine similarity) between these vectors as a feature【4†】.

The CSRT service ensures all required inputs for the model/algorithm are assembled now for each candidate.

**6. Scoring and Ranking:**

With features ready, the CSRT now evaluates each candidate through the chosen algorithm:

* **CatBoost Model Scoring:** For a ML model like CatBoost:
  + The service loads the latest trained model (either it's already loaded in memory or fetched from the model store on MapR). The model is essentially a set of decision trees and parameters that score the likelihood of the user engaging with a given merchant.
  + Each merchant's feature vector (combining user and merchant features) is fed into the model, yielding a score (often a numeric score like a propensity or relevance score). Higher score = more likely to be of interest to the user.
  + CatBoost can handle categorical features (like merchant category, user segment) and numeric features (spend amounts, distances) effectively. It might, for example, give a big weight if the merchant's category matches the user's top spend category, especially if the user has high engagement.
* **Alternative Algorithms:**
  + **Popularity:** If this path is taken (for non-personalized), the service might not do heavy feature computation. Instead, it could assign the score simply as the merchant's popularity metric (e.g., rank by number of Amex transactions or a popularity index precomputed). That list is then sorted by that score.
  + **Collaborative Filtering:** The service might compute a score based on similarity. For instance, if using user/merchant embeddings, it calculates cosine similarity between the user's preference vector and each merchant's vector (higher means more similar to the user's tastes)【4†】. Those similarity values act as the score. Or, if they have a list of merchants "people like this user also visited", it could use that directly.
  + **B2B algorithm:** Could be a custom formula or model – e.g., weight certain features differently (like give extra points to merchants in "Office Supplies" category if the user is a business card holder). The scoring might be a simpler linear combination or a decision tree model trained on business accounts' data.
  + **Random:** If truly random, the service may bypass scoring and just randomly shuffle or pick a subset of the candidates, but even then some minimal filtering applies (they wouldn't pick completely irrelevant ones, just not ordering by score).
* **Intermediate Filtering & Business Rules:** After initial scoring, the service might apply additional business logic:
  + For example, diversification: ensure the top results aren't almost identical. If the top 5 all turned out to be coffee shops, it might decide to replace a couple with other categories (if the channel's policy is to show variety). This can be done by category-based post-processing rules.
  + **Rule-based Boosts/Blocks:** If the business has priorities (say, boosting merchants that are partners or blocking ones with certain issues), these rules could adjust the scores or remove candidates at this stage.
  + **Offers integration:** If the channel is supposed to show actual offers (e.g., in a marketing email), the service can call an Offers microservice (EOS) to check if any of the top merchants have an active offer for that user【4†】. If yes, it might highlight those or even require that only merchants with offers are shown. (For a general recommendation, this step might be skipped, but it's available for specific use-cases.)

**7. Select Top N & Prepare Response:**

Now the service sorts the merchants by the final score (highest first) and selects the top N required (N might be determined by the channel – e.g., 5 for an email, 10 for a map view, etc.).

* It assembles the response payload with these top recommendations. Typically this includes the merchant identifiers, and possibly some metadata like the score or a label (the channel might simply receive IDs and fetch full details like name/address from another API or cache, or the service might include friendly names, depending on implementation).
* The CSRT ensures the format is JSON or similar, as expected by the API.

**8. Logging and Monitoring:**

As the final part of processing this request, the CSRT service logs details about the event for offline analysis and monitoring:

* **Event Logs:** A structured log entry is created containing key information – e.g., timestamp, Card Member ID (often hashed or tokenized for privacy in logs), which channel and request parameters were given, how many candidates were considered, which algorithm was used, and the final recommended merchant IDs with their scores【6†】. This log is sent to a central logging system or message queue.
* **Ensemble/Algorithm Logging:** In more advanced cases, if multiple algorithms were evaluated (say for experimentation, or if the system runs a champion vs challenger model in parallel), the results of each algorithm can be logged as well. For example, if both the CatBoost model and a collaborative filter scored the candidates (even though only CatBoost's result was returned), the service logs both sets of scores. These logs feed into an ensemble training pipeline or at least help data scientists compare algorithms later. The logs are published to a Kafka topic (or MapR Streams topic) dedicated to recommendation events【6†】.
* **Real-Time Monitoring:** Some data from the logs might also be sent to monitoring dashboards. For instance, the service could emit metrics like "recommendation API latency" or "model used = CatBoost count" to a monitoring system. Security scanners like Qualys may also continuously monitor the service and its environment, ensuring no vulnerabilities in the running application (this is more on the DevOps side, but mentioned as part of monitoring).

**9. Response to API:**

After scoring and logging, the CSRT service returns the response back to the API layer. This response includes the ranked list of recommended merchants (with any required info). The service execution for this request is complete at this point, having taken perhaps only a few hundred milliseconds thanks to the prepared data and efficient in-memory scoring.

**Batch Layer (Offline Data Pipeline)**

Behind the scenes, the Batch layer is continuously working to prepare the data and models that make the real-time CSRT service intelligent. This layer encompasses the data engineering and data science pipelines that transform raw data into features, train the ML models, and monitor their performance. We will outline its major components and how they feed the CSRT:

**(1) Data Extraction:**

The first stage in the batch pipeline is extracting and aggregating data from the Cornerstone sources:

**Card Member Data Assembly:** The pipeline pulls together various data about customers:

* **Transaction History:** All relevant transactions (purchases) made by Card Members, typically over a certain lookback window (e.g., last 12 months), from the enterprise data warehouse. This gives insight into where each customer spends (merchant, amount, date).
* **Clickstream / Engagement Data:** Logs from digital channels – e.g., how the customer has interacted with Amex's web and mobile apps. This could include page views on Amex Maps, clicks on recommended merchants or offers, email opens/clicks, etc. These come from web analytics or log systems.
* **Risk and Demographics:** The customer's risk score, credit attributes, tenure with Amex, products owned, age group, income bracket (if available), etc., often sourced from a CRMD system or customer master database.
* **Marketing Impressions/Responses:** Data on what offers or recommendations the customer has been shown in the past and whether they responded (e.g., an "impression log" and a "response log"). This helps avoid recommending the same thing too often and measures effectiveness.

These raw data pieces might reside in large tables (for example: TRANSACTIONS, WEB\_CLICKS, CUSTOMER\_PROFILE) on a Hadoop cluster or relational data store, and the pipeline will run extract jobs (SQL queries, MapReduce, Spark jobs) to gather the needed fields.

**Merchant Data Assembly:** In parallel, the pipeline gathers data about merchants:

* **Merchant Master Data:** A complete list of merchant establishments (from an internal merchant database). This includes each merchant's name, unique ID(s), industry category (MCC code or a proprietary category), location details (address, city, country, coordinates).
* **Active Merchants:** A filtered list of merchants that are "active" – for example, those that had at least one transaction from any Amex cardmember in the past X months. This helps focus the recommendations on merchants that are currently relevant. The pipeline creates this list by scanning recent transactions.
* **Merchant Attributes:** Additional attributes like whether the merchant is part of special programs (e.g., "Shop Small" eligible small business, or partnered merchant), merchant size (small/medium business vs large chain), and quality metrics (perhaps a flag if we have high confidence in the merchant's geolocation and details).
* **Merchant Linkages:** Data linking merchants to other entities – for example, grouping multiple locations of the same merchant brand, or mapping to an industry cluster or a merchant segment. There might be tables for merchant clusters (groups of similar merchants based on spend patterns or category), which will be used for features.

The pipeline likely interfaces with systems like GMS (Global Merchant System) or CRM systems that store merchant info. It may produce intermediate tables like an "Active\_Merchant\_List" and enriched merchant reference tables.

**Data Storage:** After extraction, this data is staged in the big data environment (HDFS/MapR). For instance, there could be raw fact tables on HDFS like cardmember\_transactions\_fact, merchant\_master\_dim, web\_clicks\_fact, etc., or the data may be loaded into Apache Hive tables for easier querying.

**(2) Data Roll-up & Feature Engineering:**

In this stage, the pipeline crunches the extracted data to create features at the granularity needed for the model (usually one row per entity, such as per Card Member or per Merchant):

**Card Member Feature Roll-up【6†】:** The pipeline aggregates each customer's behavior into a feature vector. Examples of Card Member features:

* **Spending Habits:** e.g., total spend in each major category (travel, dining, retail, etc.) over the last 12 months; number of transactions; favorite merchant or category (the category with highest spend count).
* **Recency/Frequency:** how recently and frequently the customer uses their card, and how often they engage with Amex digital services. E.g., "used Amex Maps 5 times in last month", "clicked on 3 merchant recommendations in the last quarter".
* **Demographics & Profile:** flags or indices like age range, income level (if known), tenure (years as a cardmember), product type (Platinum card, Corporate card, etc.), risk score band, and whether they are a consumer or small business account.
* **Engagement with Offers:** number of Amex Offers added or redeemed, previous personalized offer outcomes (did they ever act on past recommendations?), etc.
* **Geographic:** home city or ZIP (to understand local preferences), presence of travel (if they spend outside home region often – indicating if they might like travel-related recs).

These are computed using SQL or Spark jobs grouping by customer ID and joining multiple sources. The result might be stored in a table (for example, a "Customer\_Feature" table where each row is a cardmember ID with dozens of feature columns).

**Merchant Feature Roll-up【6†】:** Similarly, the pipeline computes features for each merchant or merchant-location:

* **Popularity & Usage:** e.g., number of unique cardmembers who transacted at the merchant in last 90 days (this could be the "merchant popularity score"), total transaction volume, growth trends in usage (is it becoming more popular?).
* **Customer Segments:** breakdown of what types of customers go there. For instance, 30% business cardmembers vs 70% consumer, or it's frequented by millennial-age customers – such insights can later help match customer profiles to merchants.
* **Merchant Category & Clusters:** a merchant may belong to a primary category (say, "Restaurant > Italian"). The pipeline might assign cluster IDs to merchants (grouping similar merchants together). E.g., all Italian restaurants in NYC might be cluster #110. Also, whether the merchant is in a special cluster like "airport merchants" or "e-commerce only".
* **Online Interaction Data:** if available, how often this merchant appears in search results or is clicked on Amex Maps or saved in offers. E.g., "Impression count in last month = 50, click count = 5" across all users.
* **Offer/Marketing Data:** whether the merchant is currently part of a campaign (like offering a discount through Amex Offers), which might be used to boost it when relevant.

These features are computed by aggregating transaction tables by merchant, joining with merchant attributes. The result is a "Merchant\_Feature" table keyed by merchant ID.

**Note:** In some cases, features might be rolled up at the level of merchant-location (each physical store) if recommendations distinguish between locations, or at brand level (all locations of a chain combined) depending on how the system is designed. The diagrams reference location ID level aggregation【6†】, implying location-specific data (important for geo-context).

**Feature Aggregation for Scoring:** There might be an additional step to make real-time scoring easier: e.g., pre-join certain customer and merchant features into a single structure. However, since the combination is huge (millions of customers x millions of merchants), it's more typical to keep them separate. Instead, the pipeline might ensure some indices or lookup keys are in place so that the CSRT service can quickly retrieve by ID.

For example, the features could be stored in key-value form in a NoSQL database or as flat files partitioned by ID. MapR-DB (which is JSON document DB) or HBase could be used to serve feature lookups. Or the system could simply keep them as HDFS files and load into memory for the service (depending on scale).

The output of this stage is the set of feature datasets that directly feed the model and the CSRT service. These are kept in the data lake (which is accessible by CSRT via NFS or other mechanisms).

**(3) Feature Quality Monitoring:**

As features are produced, an automated quality check runs. This is crucial in a big data pipeline because any upstream data issue (like a dropped data source or a bug in aggregation) could silently corrupt the features and thus wreck the model's effectiveness.

The Feature Quality Pipeline (possibly orchestrated by a tool, or custom scripts) checks things like:

* **Distribution changes:** e.g., the average number of transactions per customer this week vs last week – a drastic drop might indicate missing data.
* **Completeness:** ensure each expected feature column is populated and within reasonable bounds (no nulls where there shouldn't be, values like spend not negative, etc.).
* **Schema changes:** if a source table changed and a field is missing or renamed, catch it early.

The mention of CDIT/Qualys【6†】 indicates tools in place:

* **CDIT** could stand for Continuous Data Integrity Testing – an internal framework that automatically validates data quality for critical features and perhaps scores them on quality.
* **Qualys** is generally a security tool, but here it might be used in a broader monitoring context (possibly ensuring the environment and data are secure and compliant). It might also check that sensitive data in the pipeline (PII) is encrypted or tokenized properly.

If any issues are found, alerts would be raised to engineers to fix the data before it affects the model scoring.

**(4) Model Training & Monitoring:**

With fresh features generated, the Data Science team (or an automated pipeline) periodically retrains the recommendation models:

**The Model Training pipeline** takes the aggregated data (customers, merchants, and outcomes) to learn new patterns. In our case, the primary model is a CatBoost gradient boosting model. Retraining might occur, say, monthly or when a significant amount of new data has accumulated.

* **Training Data Preparation:** The pipeline assembles training examples, which likely are historical records of whether a Card Member engaged with a merchant. For example, it could treat "customer made a purchase at merchant X" or "clicked on merchant X in the app" as a positive outcome to predict, and other merchant options as negatives. It will use the features at the time of those events to train the model.
* **Grid Search & Hyperparameter Tuning【6†】:** The team may run multiple training jobs with different model hyperparameters (like number of trees, depth, learning rate for CatBoost) to optimize performance. This is done offline using a subset of data and cross-validation (e.g., K-Fold validation).
* **Model Evaluation:** They evaluate each model variant on hold-out data or via metrics like AUC (Area Under Curve), precision@N (how good the top-N recommendations are), etc. The best performing configuration is selected.
* **Champion vs Challenger:** The pipeline supports deploying a new model as a "challenger" while the current production model is the "champion". The new model might first be tested on a small percentage of traffic (via the control cell mechanism). If it consistently outperforms the champion (e.g. higher engagement rate), it can be promoted to become the new champion model.

The architecture diagram shows a **Model Monitoring Pipeline – Champion vs Challenger process【6†】**, which automates this comparison. Live performance data (clicks, conversions from recommendations) is fed back to evaluate how the challenger is doing against the baseline.

**Model Deployment:**

* Once a model is approved for use, the model artifact (e.g., a CatBoost model file containing all the trees and weights) is saved to the Cornerstone storage (for example, on the MapR file system) and possibly versioned.
* The CSRT service either periodically checks for new model files or is notified to load the latest model. Because MapR supports NFS, the CSRT service can simply load the model from a known path (e.g., a directory for "Janus\_model\_US.dat" for the latest US model). Some systems might copy the model into a low-latency KV store or even memory, but given CatBoost's format, loading from a distributed filesystem is common.
* The "Model Logs"【6†】 (from training) are also stored, which include training stats, feature importances, etc., for record-keeping.

**Continuous Monitoring:**

* Even after deployment, the Batch layer (or a connected monitoring system) tracks model performance in production. It takes the event logs of what was recommended and whether users engaged, to calculate metrics like click-through rate or conversion rate for the model. If these metrics drift or drop, it signals potential issues (either data shift or a problem with the model).
* Additionally, Qualys or similar security monitors ensure the code and environment running the model remain secure (this might be part of IT governance rather than model performance, but it's part of the overall monitoring ecosystem).

**(5) Batch Outputs Feeding Back to Real-Time:**

The end products of the batch layer – the feature tables, model files, and configurations – are what empower the CSRT service. When the CSRT service at runtime accesses a feature for a user or merchant, it's reading the output of last night's batch job. When it loads a model to score, it's using the model file trained and exported from the batch process.

The batch layer also consumes the results of the real-time layer: those event logs streamed via Kafka (CSRT's logging of shown recommendations, etc.) are ingested into Cornerstone. This closes the loop by adding to the "Impression" and "Response" data that will be used in the next training cycle or for reporting on recommendation effectiveness.

In summary, there is a continuous feedback loop: data → features → model → recommendations → new data.... The Batch layer is the factory that keeps this loop running with fresh intelligence.

**Cornerstone Layer (Data Foundation)**

The Cornerstone layer underpins the entire system with robust data storage and access frameworks. It refers to both the data infrastructure (e.g., databases, Hadoop clusters, Kafka streams) and the base datasets (raw tables) that feed the batch process. Key aspects of Cornerstone in this context:

**Enterprise Data Lake / Warehouse:** American Express's enterprise data (customer info, transactions, merchant info, etc.) lives here. In this architecture, a MapR cluster is indicated【2†】【6†】:

* MapR is a Hadoop-compatible data platform which likely stores the large tables. It allows data to be accessed via Hadoop APIs or as a mounted filesystem (NFS), which the batch jobs and even the CSRT service leverage.
* For instance, large tables like CRMD\_Customer or Transaction\_Fact might be accessible as Hive tables on MapR. The feature tables and model files are also stored on this cluster.
* MapR (or similar Hadoop FS) provides the scalability to handle billions of transaction records and the high-throughput needed for batch computations.

**Base Data Tables:** Some foundational datasets in Cornerstone used for recommendations include:

* **Customer Master (CRMD):** A comprehensive Card Member database. It provides each customer's profile: basic personal data, product holdings, segmentations, contact permissions, etc. For marketing/recommendation purposes, CRMD is often the source of demographic and lifecycle segments (e.g., flags for "Millennial Segment" or "High Value Customer").
* **Transaction Data:** All card swipe records, which are essential for computing spend-based features and merchant popularity. This likely includes fields like merchant ID, amount, date, cardmember ID, MCC code (merchant category code), etc. This dataset can be massive; typically stored partitioned by date.
* **Merchant Master Data:** The database of all merchants (could be internally called GMS – Global Merchant System). It links merchant identifiers to real-world info:
  + e.g., Merchant "Luigi's Trattoria" might have a Merchant ID, plus an acquiring reference, an address, latitude/longitude, a category code (MCC 5812 for restaurants), etc.
  + There might be separate tables for merchant locations versus merchant corporate entities. The recommendation system likely works at the location level for granularity.
* **Merchant Category/Industry Tables:** Lookup tables that map MCC codes or merchant IDs to higher-level categories (like mapping 5812->"Restaurant -> Dining" hierarchy) and clusters (grouping similar merchants for collaborative filtering or diversification logic).
* **Special Program Flags:** e.g., a table of ShopSmall merchants, or merchants enrolled in a particular promotion, used by business rules.
* **Digital Interaction Logs:** Raw logs of user interactions (web events, mobile events). These might be stored in something like a Hive table of clicks/impressions with fields [user, timestamp, action\_type, merchant\_id (if applicable), channel]. They feed into calculating features like "impression count" or "last active time on app".
* **Offer & Campaign Data:** If relevant, data about marketing offers, enrollment, etc., could be part of Cornerstone. For instance, an "Offers" table might list which Card Member has which offers available or added, and an "Offer Redemption" table for uses. This can overlap with recommendations if the strategy is to cross-promote offers.

**Streaming and Indexing Infrastructure:**

* **Kafka Topics / CS Streams:** The architecture shows Kafka as part of Cornerstone【2†】【6†】. Kafka (or MapR Streams) is used to stream data:
  + The CSRT service's logs of recommendation events are written to a Kafka topic (e.g., recs\_shown\_events). A consumer job on the batch side will read this and append to a Cornerstone table for persistent storage (like an "impression log" table).
  + Similarly, real-time data like live transactions might be flowing into Kafka topics which then feed to the data lake. This ensures the data lake is updated promptly with new data (e.g., near-real-time transaction updates could refresh popularity scores daily).
* **ElasticSearch Index:** The merchant search index is a critical piece built from Cornerstone data【2†】. There is likely a batch job that takes the merchant master data (and maybe augmented with popularity metrics) and indexes it into ElasticSearch:
  + This index allows the CSRT service to quickly retrieve merchants by location or keyword. It might be updated periodically (say daily or hourly) with any new merchants or changes (e.g., if a merchant's status changes to inactive, it can be reflected).
  + The Elastic index might have a subset of fields needed for searching and quick filtering (merchant name, address, coordinates, category tags, popularity score).
  + By using Elastic's geo-distance queries and full-text search, the system avoids heavy database queries in real-time, achieving low latency for candidate retrieval.

**Cornerstone as Single Source of Truth:** The Cornerstone layer ensures that the recommendation system uses consistent and trusted data:

* All the metrics and features ultimately trace back to these base tables. For example, if a merchant's popularity score is high, that's because the transaction table in Cornerstone showed many transactions for it. If a user is labeled "Foodie", that's derived from their transaction data in Cornerstone.
* This layer also handles data governance – ensuring privacy (data is tokenized where needed, access is controlled), and accuracy (via the monitoring processes).
* The Cornerstone data is often shared across many systems, not just recommendations. For instance, CRMD or transaction tables are used by finance, risk, etc. The recommendation pipeline is one consumer, which means it inherits the robustness (and sometimes the limitations) of these enterprise datasets.

**End-to-End Flow Illustrated**

Bringing it all together, here's a cohesive example of how a personalized recommendation is delivered to a Card Member through these layers:

**Step A: User Interaction (Channel)** – A Card Member launches the Amex mobile app and opens the Amex Maps feature. The app automatically wants to show 5 "Recommended for You" merchants on the map. It obtains the user's current location (say, Midtown Manhattan) via GPS. Because the user is logged in on the app, the app knows their Card Member ID. It sends a request to the backend API: "Give me 5 recommended merchants near [latitude=40.75, longitude=-73.99] for user [ID 12345], context=Maps."

**Step B: API Processing** – The Recommendation API receives this request. It verifies the user's auth token and identifies this as a secured, logged-in request. Based on the global configuration, the API knows that logged-in Maps requests should be personalized. (For instance, a config entry might say: channel=Maps, experiment=PersonalizedModel vs Popularity, user 12345 is not in the hold-out group, so go with PersonalizedModel.) The API prepares a call to the CSRT service, including all relevant details (userID, location coordinates, request for 5 recommendations, channel=Maps). It does not yet decide which merchants – that's up to CSRT – but it might include a hint like "algo=ML" if the config is external.

**Step C: Real-Time Scoring (CSRT)** – The CSRT service starts handling the request:

* It looks up user 12345's precomputed feature profile from the feature store on MapR. This provides dozens of attributes (e.g., user 12345 is a Platinum cardmember, spends 30% on dining, high travel spend, in NYC area, loves Italian cuisine, etc.).
* It determines via its router logic that this request should use the CatBoost ML model (since personalization is enabled and this user isn't in a random control cell).
* The service converts the provided GPS coordinates to a suitable search query. It might round them or determine the bounding box for a search (e.g., 5-mile radius).
* CSRT queries the ElasticSearch merchant index with a geo query: "find merchants within 5 miles of 40.75,-73.99". It also might filter for merchant categories that match the user's interests – but since the user didn't explicitly filter (they just opened the map), it could retrieve a broad set of merchants, letting the model decide which are interesting. Elastic returns, say, 100 merchants in midtown Manhattan that are known to Amex.
* The service filters these candidates: it drops any merchants that are not in the "active merchant" list or that lack enough data for scoring. It ensures a variety of categories are present (maybe it limits to at most 50% of one category in the candidate list to give the model a mix).
* CSRT then fetches merchant features for each of the remaining candidates. For example, one candidate is "Joe's Pizza on 9th Ave" – features say it's a restaurant (pizza), very popular (in top 20% by transactions), small business flag = true. Another is "Starbucks on 8th Ave" – category coffee shop, popular chain (popularity high, but not a small business).
* Now, for each merchant, CSRT combines the merchant's features with user 12345's features and runs the CatBoost model:
  + The model might output a higher score for "Joe's Pizza" because the user's profile indicates a love for dining and especially non-chain local eateries, plus Joe's is within 1 mile. It outputs a slightly lower score for "Starbucks" because even though the user often buys coffee, the model has learned that this user prefers unique places over chains (hypothetically).
  + It scores all candidates similarly.
* CSRT sorts the merchants by score and takes the top 5. Suppose the top recommendations (with scores) are: 1) Luigi's Trattoria (score 0.92), 2) Joe's Pizza (0.85), 3) Blue Bottle Coffee (0.80), 4) Shake Shack (0.75), 5) Starbucks (0.60).
* It notices the top two are both Italian restaurants (Luigi's and Joe's). A business rule for Maps might say not to show two very similar cuisines at once. So it decides to swap out Joe's Pizza (rank 2) with the next one down that's a different category (Blue Bottle Coffee, which is rank 3). Now Joe's Pizza will be rank 3 and Blue Bottle rank 2 in the final list, ensuring a mix of dining and coffee.
* The service packages these 5 merchants into a response payload, including basic info like merchant ID, name, maybe category or a tagline (it could also just send IDs and let the front-end or another API get names).
* **Logging:** Before sending, CSRT logs this event:
  + It creates an entry: User 12345, request time, location given, algorithm=CatBoost, candidates considered=100, top5 merchants = [Luigi's, Joe's, Blue Bottle, Shake Shack, Starbucks] with their scores.
  + This log is pushed to the Kafka topic for recommendation events. It will be consumed later by the data pipeline to record that these merchants were shown to user 12345 on this date.
  + Additionally, it might log that model version "Janus\_US\_v5" was used. And since a challenger model is also running for 5% of traffic (not this user though), if that were relevant it would log something about that too.

**Step D: API Response to Channel** – The Recommendation API receives the response from CSRT. It passes the data back to the Amex Maps front-end. Depending on implementation, the API might do minor formatting (or even caching if multiple calls come quickly for the same area, though personalization makes caching less useful).

**Step E: Channel Displays Results** – The Amex Maps app now has the 5 recommended merchants. It calls another API to fetch their details (if needed) or it might have gotten names in the payload. The app then plots these 5 locations on the map and perhaps highlights them as "Recommended for You." The user sees Luigi's Trattoria, Blue Bottle Coffee, Shake Shack, etc. as recommended spots near them.

**Step F: User Feedback Loop** – The user's interactions will feed back:

* Suppose the user clicks on Luigi's Trattoria and views details – that click is captured in the app's analytics and will be sent to the backend (perhaps via a separate clickstream event).
* If the user actually goes there and uses the Amex card, that transaction will flow into the data warehouse.
* All these signals (the impression, the click, and later the transaction) will be linked to the fact that Luigi's was recommended. In the next batch cycle, the pipeline may mark this as a successful recommendation for model training (positive label).
* Over time, this helps the model learn which recommendations led to engagement, closing the learning loop.

Finally, it's worth noting how each layer contributed in this flow:

* **Channel** provided context and a user interface for the recommendations.
* **API** handled authentication, routing, and ensured the request was directed properly with the right parameters.
* **CSRT microservice** did the heavy lifting: retrieving data, applying the ML model and rules, and producing the tailored recommendation list in real-time.
* **Batch layer** had pre-processed all the necessary data (features, model) that CSRT used. Without fresh features or a trained model, CSRT would have nothing smart to work with.
* **Cornerstone** was the solid foundation, storing all the raw data and serving as the source of truth for both the batch computations and real-time queries (via Elastic and lookup tables). It also captured the outcome, which will be used to refine future recommendations.

Through this coordinated five-layer architecture, the system ensures that each Card Member sees relevant, personalized merchant recommendations, delivered quickly and based on up-to-date data and models. Each layer interacts seamlessly with the next – from the moment the user triggers a request, to the data-driven computation in the backend, and back to the user with a set of recommendations – thereby closing the loop between data, algorithms, and user experience.