# Architectural Blueprint for a Marketing Mix Modeling Simulation Engine

## Section 1: Deconstructing the Marketing Simulation Engine: Core Domain Models

The successful development of a sophisticated simulation game hinges on a robust and accurate underlying model of the domain it seeks to represent. In the context of a Marketing Mix Modeling (MMM) simulation, this requires translating complex statistical and marketing concepts into a concrete, deterministic, and testable set of data structures and functions. This section outlines the foundational domain models for the simulation engine, establishing a core built upon pure, composable functions. This architectural approach ensures that the simulation's logic is entirely decoupled from the user interface and state management layers, forming a solid, predictable foundation for the entire application.

### 1.1 The Foundational Loop: Modeling the Marketing Mix (MMM)

The core of the simulation must replicate the fundamental process of Marketing Mix Modeling. MMM is a time-tested analytical technique that uses statistical methods, primarily multi-linear regression, to measure the impact of various marketing investments on business performance.1 The game's primary loop will simulate this process over discrete time steps, such as weeks or months, allowing the player to make decisions and observe their impact.

The central principle of MMM is the decomposition of total sales into its constituent parts. The simulation's primary output for any given time step will be the separation of total sales into baseSales and incrementalSales.2 BaseSales represents the baseline demand for a product, driven by factors independent of short-term marketing, such as brand equity, seasonality, pricing, and distribution. IncrementalSales, conversely, is the volume of sales generated directly as a result of the player's marketing activities and promotional campaigns.2 This relationship forms the central equation of the game engine for each time step, t:

The player's primary input into this model will be the allocation of a budget across various marketing channels (e.g., TV Advertising, Print, Digital Marketing, Social Media). However, a realistic model must also account for a variety of non-media drivers and external factors that influence sales.1 These can include competitor activity, seasonality, macroeconomic trends like inflation, and specific events like holidays.1 Within the game, these will serve as parameters for the simulation engine; some will be directly controlled by the player (budgets, promotions), while others will be dictated by the game's scenario engine to create dynamic challenges.

The primary dependent variable of the model is, naturally, sales volume or value. From this, the simulation will derive a suite of Key Performance Indicators (KPIs) to provide the player with actionable feedback. These outputs are crucial for strategic decision-making and include channel-specific contribution to sales, effectiveness (sales generated per unit of marketing effort), efficiency (sales generated per dollar spent), and, most critically, Return on Investment (ROI) for each channel.2

A fundamental architectural decision for the simulation engine is to model its core logic as a pure function. A pure function is one whose return value is determined solely by its input arguments, without any observable side effects like modifying external state or performing I/O operations.6 A simulation, by its nature, must be deterministic: given an identical set of inputs (player decisions and market conditions), it must always produce the exact same output.8 This requirement aligns perfectly with the definition of a pure function.

Therefore, the entire simulation logic for a single time step can be encapsulated within a single, top-level pure function signature: runSimulationTick(previousState, playerInputs, marketConditions): newState. This design choice yields significant architectural benefits. It makes the core engine eminently testable, as its behavior can be verified by asserting outputs against a known set of inputs without any complex setup or mocking.7 It enables features like "undo" functionality or scenario replays simply by storing a history of inputs and re-running the simulation. Most importantly, it creates a clean separation of concerns, completely decoupling the complex business logic of the engine from the state management layer (Zustand) and the rendering layer (React).

To formalize this structure, the data contracts for the entire application must be explicitly defined using TypeScript interfaces. This is the most critical first step in migrating from an untyped JavaScript prototype, as it eliminates ambiguity and provides a clear blueprint for all development.9 The following table outlines the core data models.

| Interface/Type Name | Property | TypeScript Type | Description & Source Snippet |
| --- | --- | --- | --- |
| Channel | - | enum | Defines the set of available marketing channels (e.g., TV, Radio, PaidSearch). Using an enum enhances type safety and code clarity.11 |
| PlayerInput | channelBudgets | Record<Channel, number> | The player's primary decision: how much budget to allocate to each marketing channel for the current time step.1 |
|  | promotions | Promotion | An array of promotional activities planned for the current time step, such as pricing discounts.1 |
| MarketConditions | seasonalityIndex | number | A multiplier (e.g., 1.2 for high season) that affects base sales, representing a non-media driver.1 |
|  | competitorSpend | Record<Channel, number> | The level of spending by competitors in each channel, which can impact the effectiveness of the player's own spend.1 |
|  | economicIndex | number | A value representing macroeconomic factors like GDP or inflation that can influence overall consumer purchasing power.1 |
| SimulationState | tick | number | The current time step of the simulation (e.g., week number). |
|  | marketConditions | MarketConditions | The market conditions for the current tick. |
|  | adstock | Record<Channel, number> | The carried-over advertising effect from previous periods for each channel. This is a critical piece of state that must persist between ticks.12 |
|  | results | SimulationOutput | The calculated KPIs and outcomes for the current tick. |
| SimulationOutput | totalSales | number | The total sales value or volume generated in the current tick.2 |
|  | baseSales | number | The portion of total sales attributed to baseline factors, independent of short-term marketing.2 |
|  | incrementalSales | number | The portion of total sales attributed directly to the player's marketing efforts.2 |
|  | channelContributions | Record<Channel, number> | The amount of incremental sales attributed to each individual marketing channel.2 |
|  | channelRoi | Record<Channel, number> | The calculated Return on Investment for each channel, a primary KPI for the player.1 |

### 1.2 Modeling Temporal Dynamics: The Adstock Effect

A critical component of a realistic marketing simulation is the concept of "Adstock," also known as the carryover effect. This phenomenon describes how the influence of advertising on consumer behavior persists long after the initial exposure, decaying gradually over time rather than disappearing instantly.12 This introduces a crucial element of memory into the simulation: the outcomes of the current period are dependent not only on current actions but also on the history of past actions.

The most widely used mathematical model for this effect is the geometric decay adstock. It is defined by a simple but powerful recursive formula that calculates the adstock level at a given time t based on the marketing spend in that period and the adstock level from the previous period, t-1.12

In this formula, the key parameter is the decay rate, represented by the Greek letter alpha (). This value, which must be between 0 and 1, determines the speed at which the advertising effect fades. A high decay rate (e.g., 0.8) means that 80% of the previous period's adstock carries over, indicating a long-lasting effect. A low decay rate (e.g., 0.2) signifies a rapid drop-off.12 The concept of "half-life"—the time it takes for the adstock to decay to half its value—can be mathematically derived from the decay rate and often provides a more intuitive metric for players to understand the persistence of their marketing efforts.13

The decay rate is not a universal constant; it varies significantly between different marketing channels. For instance, a highly creative and memorable television campaign may have a very slow decay rate, its message lingering in the minds of consumers for weeks. In contrast, a transient digital banner ad might have a very fast decay, its impact fading almost immediately after the impression is gone.12 Consequently, the simulation's Adstock model must be parameterized on a per-channel basis, with each channel having its own distinct decay rate.

The recursive nature of the Adstock formula has a direct implication for the simulation's state management. The calculation Adstock\_t depends on Adstock\_{t-1}, which means the adstock value for each channel is a piece of state that must be carried over from one simulation tick to the next. This reinforces the design of the main simulation function. The runSimulationTick function must accept the previousState.adstock values as part of its input and return the newly calculated newState.adstock values as part of its output. The core logic can be implemented as a pure function, calculateAdstock(currentSpend, previousAdstock, decayRate): newAdstock, which takes the necessary inputs and returns the new value. The responsibility of managing and persisting this state between ticks is handled at a higher level by the main loop, preserving the purity and testability of the core calculation.

### 1.3 Modeling Investment Efficiency: Saturation and Diminishing Returns

Applying the adstocked spend value directly to sales in a linear fashion would be an unrealistic simplification. In reality, marketing investments exhibit the economic principle of diminishing returns. The first dollar spent on a campaign, which reaches a fresh and receptive audience, is far more effective than the millionth dollar, which may reach an already saturated audience or generate only marginal additional awareness.15 This phenomenon is known as the saturation effect.

To accurately model this non-linear relationship, the industry standard is the Hill function, which generates a characteristic S-shaped response curve.15 This function effectively captures the dynamics of marketing response: an initial phase of low impact at low spending levels, followed by a steep increase in effectiveness during a growth phase, and finally, a plateau where additional spending yields progressively smaller gains.17 The Hill function can be expressed mathematically as:

Here, x represents the input variable, which in our simulation would be the adstocked spend for a given channel. The shape of the curve is controlled by two key parameters:

* ec (also referred to as $K\_d$ or the half-saturation point): This parameter represents the level of spend at which the response reaches 50% of its maximum potential. It is a critical indicator of a channel's efficiency and the point at which diminishing returns begin to dominate.15
* slope (or $n$, the Hill coefficient): This parameter controls the steepness of the S-curve. A higher slope indicates a more rapid transition from the low-impact phase to the saturation phase, while a lower slope represents a more gradual response.15

Similar to the Adstock decay rate, these saturation parameters are unique to each marketing channel. A channel like targeted paid search might have a low half-saturation point and a steep slope, saturating very quickly. Conversely, a broad-reach channel like national television advertising might have a very high half-saturation point and a gentler slope, capable of absorbing a much larger budget before its effectiveness plateaus.16 These parameters must be configured individually for each channel within the simulation's design to create strategic differentiation.

An important and subtle modeling decision is the order in which the Adstock and Saturation transformations are applied. The transformations can be composed in two ways: Adstock(Saturation(Spend)) or Saturation(Adstock(Spend)).15 This is not merely a technical detail but a choice with significant strategic implications for the simulation's behavior.

1. **Saturation before Adstock (Adstock(Saturation(Spend))):** This sequence implies that the diminishing returns effect occurs at the moment of spending. The already-diminished impact of that spend is then what contributes to the consumer's memory (Adstock).
2. **Saturation after Adstock (Saturation(Adstock(Spend))):** This sequence suggests that the raw marketing spend builds up in the collective consumer consciousness first. The saturation effect then occurs on this total accumulated "mental share" or awareness.

These two approaches will yield different results, particularly for channels with high decay rates and varying spend patterns. This modeling choice can be a fixed characteristic of certain channels to add realism, or it could even be exposed to the player as an advanced strategic lever, adding another layer of depth and complexity to the gameplay.

### 1.4 Modeling Systemic Complexity: Cross-Channel Synergy and Interaction

Marketing channels do not operate in isolation; they exist within a complex ecosystem where they can influence one another's effectiveness. These interactions can be positive, creating a synergistic effect where the combined impact is greater than the sum of the parts (a "2 + 2 = 5" scenario).18 Conversely, they can be negative, leading to cannibalization or interference where the combined effect is less than the sum of the parts (a "2 + 2 = 3" scenario).18 A sophisticated and realistic simulation must account for these crucial cross-channel dynamics.

In statistical regression, these relationships are modeled using an interaction term. For two independent variables, $X\_1$ and $X\_2$, the equation would be:

Here, the $\beta\_3$ coefficient captures the interaction effect.21

* **Positive Interaction (Synergy):** If $\beta\_3$ is positive, the two channels are synergistic. For example, a television ad campaign ($X\_1$) might build broad brand awareness. When a consumer who has seen the TV ad later sees a targeted digital ad ($X\_2$), they are more likely to click and convert. The two campaigns working in tandem produce more sales than if they had been run independently.21
* **Negative Interaction (Cannibalization):** If $\beta\_3$ is negative, the channels interfere with each other. This can happen when two similar digital campaigns target overlapping audiences, effectively competing against each other and driving up auction costs, which reduces the overall ROI.5 Another example is when a broad promotion cannibalizes sales from a more targeted one.

While modeling every possible two-way or three-way interaction with distinct regression terms is statistically sound, it can become computationally intensive and, more importantly, difficult for a player to comprehend and act upon. A more elegant and game-friendly abstraction is required to represent this complexity.

A powerful way to model these relationships is through a **Synergy Matrix**. While the concept of a synergy matrix is often used in a general business context for aligning teams like marketing and sales 22, it can be adapted specifically to model the interactions between marketing channels. This would be implemented as an N x N matrix, where N is the number of marketing channels in the simulation. The value at Matrix[i][j] would be a multiplier that represents the interaction effect between channel i and channel j.

* A value greater than 1.0 (e.g., 1.1) would indicate a 10% synergistic boost.
* A value less than 1.0 (e.g., 0.9) would indicate a 10% cannibalistic effect.
* A value of exactly 1.0 would indicate no interaction.

The simulation engine would first calculate the individual, adstocked, and saturated impact of each channel. Then, as a final step, it would apply the synergy multipliers from the matrix based on the player's overall mix of active channels. This approach simplifies the underlying model while capturing the essential strategic dynamic of channel interaction, presenting it to the player in a more understandable and actionable format.

## Section 2: Architectural Blueprint for Migration: From JavaScript Prototype to a Scalable TypeScript Engine

This section provides the technical execution plan for migrating the existing JavaScript prototype to a new, robust architecture. The primary goals are to establish a clean, modular, and rigorously tested codebase that leverages the full power of TypeScript for type safety and maintainability. The architecture will enforce a strict separation between the core simulation logic, the application state, and the user interface.

### 2.1 A Phased Migration Strategy for Type Safety and Maintainability

Attempting to migrate an entire codebase from JavaScript to TypeScript in a single "big bang" effort is fraught with risk. It can be highly disruptive to the development process, introduce a multitude of unforeseen bugs, and overwhelm the development team.9 A gradual, phased migration strategy is the recommended approach. It allows for incremental progress, ensures the application remains stable throughout the process, and provides the team with an opportunity to learn and adapt to the new TypeScript environment.9

The migration should be viewed not merely as a task of adding types to existing code, but as a strategic opportunity to refactor and implement the new, improved architecture defined in Section 1. Each file that is converted is a chance to enforce best practices, refactor its logic into pure functions, place it within the correct module, and establish comprehensive unit tests. This reframes the migration from a technical chore into the primary mechanism for building the target architecture, ensuring that by the end of the process, the new system is fully realized.

The migration process should follow these distinct steps:

1. **Setup and Configuration:** The first step is to establish the development environment. This involves installing TypeScript and its related testing tools, such as ts-jest. A tsconfig.json file must be created and configured with the strict: true option enabled from the outset. While permissive settings might seem easier initially, starting with strict mode enforces best practices from day one and maximizes the benefits of TypeScript's static analysis, catching more potential errors at compile time.10
2. **Define Core Types:** Before migrating any functional code, the development team should implement the core data structures defined in the table in Section 1.1. Creating these shared interfaces (PlayerInput, SimulationOutput, MarketConditions, etc.) in a central types directory provides a stable, type-safe foundation that will accelerate the migration of all functions and modules that depend on them.
3. **Gradual File Conversion:** The migration should proceed on a module-by-module basis. The process begins by renaming a .js file to .ts or .tsx. It is best to start with the innermost, dependency-free utility modules and work outwards towards more complex, dependent modules.9 This approach minimizes cascading errors and allows for a more controlled and manageable process. Critical and widely-used code should be prioritized to maximize the early benefits of type safety.10
4. **Incremental Typing:** Upon converting a file, the TypeScript compiler will likely report numerous errors. The any type can be used as a temporary "escape hatch" to silence these errors and get the file to a compilable state. However, the use of any should be strictly controlled and considered technical debt. A firm policy should be in place that a module's migration is not considered complete until all instances of any have been replaced with specific, accurate types.10
5. **Refactor to Best Practices:** As each file is migrated, it should be refactored to align with modern TypeScript best practices. This includes replacing magic strings or numbers with enums for fixed option sets (e.g., channel names, API statuses), breaking down large functions into smaller, reusable helper functions, and organizing code into a clean, modular structure.11

### 2.2 Core Engine Architecture: Embracing Pure Functions and Modularity

The core simulation engine, which contains the critical business logic, must be designed to be self-contained, highly testable, and reusable. The ideal architecture for this is one based on a modular structure of pure functions, which promotes clarity, maintainability, and isolation.6

A clear and logical directory structure is the foundation of a modular architecture. The simulation engine's code should reside in a dedicated /src/engine directory. Within this, subdirectories should be created for each core domain model, mirroring the structure laid out in Section 1: /engine/adstock, /engine/saturation, /engine/synergy, and so on.11

Each of these modules should be composed of small, single-responsibility pure functions. For example, the /engine/adstock module would export a calculateGeometricAdstock function, while the /engine/saturation module would export an applyHillTransform function. This approach, favoring granular functions over monolithic ones, significantly improves readability, as each function does one thing well. It also dramatically simplifies testing, as each small unit of logic can be validated in isolation, and enhances maintainability, as changes to one piece of logic are less likely to have unintended consequences elsewhere.24

To maintain clean boundaries and enforce encapsulation, each module directory should contain an index.ts file. This file serves as the public API for the module, explicitly exporting only the functions and types that are intended for external use while keeping internal helper functions and implementation details private to the module.26

The main simulation function, runSimulationTick, will reside at the top level of the engine module (e.g., in /engine/index.ts). Its role is to act as a composer, orchestrating the calls to the various granular functions from the sub-modules. It will take the previous state and player inputs, pass them through the adstock, saturation, and synergy functions in the correct sequence, and return the new, fully calculated state. This creates a clear hierarchy of logic.

This architecture combines two powerful concepts. Internally, the engine is highly granular and modular, which is optimal for development, testing, and maintenance. Externally, however, it can present a simple, unified interface to the rest of the application. The main index.ts can export a single SimulationEngine object that exposes only the high-level methods needed by the state management layer, such as runSimulationTick and initializeState. This provides the best of both worlds: the internal complexity is managed and encapsulated, while the external API is clean and easy to consume.

### 2.3 State Management with Zustand: Patterns for a Simulation Environment

Zustand is a minimal, unopinionated state management library that is well-suited for this project.28 To effectively integrate the complex, external simulation engine, a robust and scalable pattern is required. This pattern must ensure efficient state updates and prevent unnecessary re-renders in the React user interface, which is a common performance bottleneck in complex applications.29

For an application with highly interrelated state, as is the case with a turn-based simulation, the recommended approach is to use a single global Zustand store. This store should be logically partitioned into "slices," which are dedicated sections of the state object for different domains of data.29 This practice keeps the state organized and helps in creating targeted selectors for performance optimization.

All state updates in a React/Zustand environment must be immutable; the existing state object should never be directly mutated.28 The set function provided by Zustand handles this. For the simulation, each tick will generate a completely new simulationState object. The set function will be used to replace the old state with this new one, which is an efficient and valid pattern for managing discrete state transitions.28

The bridge between the pure simulation engine and the state management layer is the Zustand "action." An action is a function defined within the store that is responsible for modifying the state.30 A central action, such as advanceTick(playerInputs), will be created. When this action is called, it will:

1. Access the current state using the get() function provided by Zustand.
2. Call the external, pure SimulationEngine.runSimulationTick() function, passing in the current state and the new player inputs.
3. Receive the newState object returned by the engine.
4. Use the set() function to update the store with this new state.29

Zustand natively supports asynchronous actions, which is useful if the simulation needs to fetch external data, such as dynamic market conditions or competitor moves from an API, as part of its tick logic.28

A particularly powerful and clean architectural pattern involves leveraging Zustand's ability to operate without React. The createStore function from the zustand/vanilla entry point creates a plain JavaScript object with getState, setState, and subscribe methods, completely independent of any UI framework.28 This allows for a perfect decoupling of the simulation engine from the user interface.

The core simulation state can be managed by a vanilla store. The main game loop, driven by a setInterval timer, can interact directly with this vanilla store, calling its actions to advance the simulation tick, completely unaware of React. The React application can then be built on top of this. The standard React hook, create, can be initialized with the existing vanilla store instance (const useGameStore = create(vanillaStore)). This effectively makes the React components a reactive view layer that subscribes to the state of the independent, underlying simulation engine. This separation is a hallmark of a highly scalable and maintainable architecture, as the core game logic can be developed, tested, and even run in different environments (e.g., a server) without the React front-end.

The following table provides a concrete blueprint for the Zustand store's structure, organized into logical slices.

| Slice Name | State Properties (with types) | Actions | Purpose & Rationale |
| --- | --- | --- | --- |
| simulationSlice | tick: number status: 'idle' | 'running' | 'paused' marketConditions: MarketConditions history: SimulationState | advanceTick: (inputs: PlayerInput) => void reset: () => void play: () => void pause: () => void | Holds the core simulation state, including a history of past states for trend analysis and visualization. Manages the game loop and its state. |
| inputSlice | channelBudgets: Record<Channel, number> plannedPromotions: Promotion | setBudget: (channel: Channel, value: number) => void addPromotion: (promo: Promotion) => void clearInputs: () => void | Manages the player's transient inputs for the *next* tick. This state is separate from the main simulation state, allowing players to plan and adjust their decisions before committing them via the advanceTick action. |
| uiSlice | activeTab: 'dashboard' | 'planning' modal: { isOpen: boolean; content: React.ReactNode } | setActiveTab: (tab: 'dashboard' | 'planning') => void openModal: (content: React.ReactNode) => void closeModal: () => void | Manages the state of the user interface itself, such as which view is currently active or if a modal dialog is open. This state is completely separate from the simulation logic. |

### 2.4 A Robust Testing Framework for Simulation Integrity

The correctness and predictability of the simulation engine are paramount to the game's success. A multi-layered testing strategy, leveraging the Jest testing framework, is essential to guarantee the integrity of the simulation's mechanics. This strategy will encompass both fine-grained unit tests for individual logical components and broader integration tests to validate the behavior of the system as a whole.23

The initial setup requires configuring Jest to work seamlessly with TypeScript. This is achieved by installing ts-jest and creating a jest.config.js file that specifies ts-jest as the preset and transformer for .ts files. The project's tsconfig.json should also be configured to include Jest's type definitions to ensure a smooth development experience.23

Unit Testing:

The focus of unit testing is to verify the smallest possible pieces of code—individual functions—in complete isolation from the rest of the system.31 The architectural decision to build the engine from small, pure functions makes this process exceptionally effective and straightforward. Each function, such as calculateGeometricAdstock or applyHillTransform, can be tested by providing a set of known inputs and asserting that the returned output is exactly as expected. Because these functions have no external dependencies or side effects, no mocking or complex setup is required.7 The test suite for each function should cover not only the typical cases but also edge cases, such as a spend of zero, a decay rate of 1, or an input that should result in maximum saturation.32

Integration Testing:

Integration tests are designed to verify that different units of code work correctly when combined.31 Within the context of the simulation engine, the primary integration test will focus on the main runSimulationTick function. This test validates the composition of the granular units. The test would involve creating a complete initialState, a set of playerInputs, and marketConditions. It would then call runSimulationTick and perform assertions on the resulting newState object. This verifies that the Adstock, Saturation, and Synergy models are being applied in the correct order and are interacting as designed, ensuring the integrity of the simulation as a whole system.33

For developing the core mathematical components of the engine, a Test-Driven Development (TDD) approach is highly recommended. TDD is a methodology where the developer first writes an automated test case that defines a desired improvement or new function. This test will initially fail. The developer then writes the minimum amount of code necessary to make the test pass, and finally refactors the new code to acceptable standards.34

This "red-green-refactor" cycle is a perfect fit for implementing the simulation's mathematical models. For example, to implement the calculateGeometricAdstock function, one would first write a Jest test:

test('should correctly apply geometric decay', () => { expect(calculateGeometricAdstock(100, 200, 0.5)).toBe(200); });

This test, which captures the formula 100 + (0.5 \* 200), will fail because the function doesn't exist yet. The next step is to write the function's implementation to make the test pass. This process guarantees that the code implementation perfectly matches the documented mathematical models from the research and ensures that the critical engine logic has 100% test coverage from its inception.

## Section 3: Planning for Interactivity and Motion: UI/UX and Visualization

With a robust and well-tested simulation engine architecture in place, the focus shifts to the user-facing components. This section outlines the research and planning required to develop an intuitive and engaging front-end experience. It covers the design of the player's control panel, the effective visualization of complex simulation outputs, and the technical architecture for connecting these elements into a cohesive and performant application.

### 3.1 Designing the Simulation Control Panel: User Inputs and Controls

The player's ability to interact with and influence the simulation is the core of the gameplay experience. The user interface must provide clear, intuitive, and efficient controls for making strategic decisions, primarily centered around budget allocation. Research into existing educational and financial simulations shows that common UI patterns for these inputs include interactive worksheets, data entry forms, and sliders, which allow for both precise and exploratory budget adjustments.35

Beyond simple budget numbers, a comprehensive marketing simulation should allow players to make qualitative decisions that impact the model's parameters. The interface should provide controls for decisions related to brand design, advertising messaging, pricing strategies, and distribution channel selection, as these are all integral parts of the marketing mix.38

A key feature of professional MMM platforms is the ability to conduct "what-if" scenario testing.1 The game's UI should incorporate this concept to enhance strategic depth. This could be implemented as a planning mode where players can allocate a hypothetical budget, see a real-time forecast of the likely outcome based on the simulation model, and then commit those decisions to run the next time step.40

To create a clear and logical user flow, the UI should be structured around the core gameplay loop: **Analyze -> Decide -> Execute -> Observe**. Instead of presenting the player with a single, monolithic screen containing all information and controls, a more effective approach is a multi-view or tabbed interface that guides the player through this cycle.

1. **Dashboard View (Analyze/Observe):** This screen is the primary display for the results of the most recently completed simulation tick. It would feature all the key data visualizations (detailed in the next section) that allow the player to analyze their performance and understand the state of the market.
2. **Planning View (Decide):** This screen contains all the interactive controls—sliders, input fields, selection boxes—that the player uses to set their budget and make strategic decisions for the *upcoming* tick. This view is the ideal place to integrate the forecasting and "what-if" analysis tools.
3. **Execution Trigger:** A prominent call-to-action, such as a button labeled "Run Next Week," serves as the "Execute" step. Clicking this button commits the inputs from the Planning View, triggers the advanceTick action in the state store, and transitions the user back to the Dashboard View to observe the new results. This UI flow directly mirrors the strategic decision-making process, providing a clear, structured, and intuitive user experience.

### 3.2 Visualizing Simulation Outcomes: From Raw Data to Actionable Insight

The simulation engine will produce a significant volume of complex data with each tick. Raw numbers in a table are not enough; effective data visualization is critical to transform this data into actionable insights that empower the player to make informed strategic decisions.

The standard for displaying such complex data is a dashboard interface, which aggregates various charts, graphs, and key performance indicators into a single, comprehensive view.40 The choice of visualization should be tailored to the data being presented.

* **Trends Over Time:** Line graphs are ideal for showing the historical performance of metrics like total sales, brand equity, or channel-specific ROI over multiple ticks.
* **Contribution Breakdowns:** Pie charts or stacked bar charts are effective for showing the decomposition of total sales, illustrating the percentage contribution from base sales versus each marketing channel.2
* **Hierarchical Data:** Treemaps or sunburst charts can be used to visualize more complex relationships, such as the breakdown of sales by region and then by channel within each region.41
* **Key Metrics:** The UI must prominently display the core outputs of the MMM process, including clear tables or cards showing the sales contribution, ROI, and efficiency for each marketing channel.1 If the simulation includes more granular digital marketing aspects, metrics like Click-Through Rate (CTR), Cost Per Click (CPC), and Customer Acquisition Cost (CAC) should also be visualized.43

To be truly effective, these visualizations must be interactive. A player should be able to hover over a data point to see a detailed tooltip or click on a segment of a chart to "drill down" for more information, filtering other dashboard components to provide deeper context, a technique demonstrated effectively in professional analytics tools.41

Beyond visualizing the outputs, a powerful and unique feature for this game would be to visualize the "invisible" mathematical models that drive the simulation itself. The abstract concepts of Adstock and Saturation are central to the game's logic, but a player cannot optimize their strategy if they cannot build an intuitive understanding of how these models behave.

A highly effective UI feature would be a dedicated analysis screen for each marketing channel that displays its calculated **Response Curve**. This would render the S-shaped curve generated by the Hill function for that specific channel. The visualization would then plot the player's current (or planned) adstocked spend on that curve. This would instantly and visually communicate to the player their current position on the curve: whether they are in the initial phase of increasing returns, the optimal "sweet spot" of high efficiency, or the phase of diminishing returns where their budget is saturating the channel. This transforms an abstract mathematical model into a tangible, actionable, and strategic tool, elevating the gameplay from simple trial-and-error to a sophisticated exercise in optimization.

### 3.3 Architecting the Game Loop and Rendering Cycle

The final architectural consideration is to connect the simulation engine, the state management layer, and the UI components into a cohesive, running application. This involves establishing the game loop that drives the simulation forward and ensuring the UI updates efficiently in response to state changes.

The core progression of the simulation will be managed by a timer-based game loop, most commonly implemented using setInterval.44 At each interval (e.g., every 1-2 seconds), the loop will trigger the advanceTick action in the Zustand store, which in turn executes the simulation engine logic.

The React components that make up the UI will subscribe to the Zustand store. Crucially, they should not subscribe to the entire state object. Instead, each component should use a targeted selector hook to subscribe only to the specific slices of data it needs to render (e.g., const results = useGameStore(state => state.simulation.history)). This is a key performance optimization. When the state updates after a simulation tick, Zustand will perform a strict equality check on the selected state, and only the components whose selected data has actually changed will re-render.28

While the core simulation is data-driven, the user experience can be greatly enhanced by the thoughtful use of animation and motion. Abrupt changes in numbers and charts can be jarring. Libraries such as Framer Motion or React Spring can be integrated to create smooth transitions when data changes. For example, as bar charts update, the bars can animate to their new heights, and numerical KPIs can animate a "count up" effect. These animations are triggered by the state changes from Zustand, creating a polished and responsive feel.

A critical architectural principle for performant and stable game development is the separation of the **simulation tick** from the **render tick**. The simulation tick is the rate at which the game's state is updated; this should be a fixed interval managed by setInterval. The render tick is the rate at which the screen is drawn, which is managed by the browser via requestAnimationFrame and can be variable. The proposed architecture, which uses a vanilla Zustand store for the engine, naturally enforces this separation. The simulation can "tick" at a consistent rate of once per second, updating the core state. The React rendering cycle runs independently and will simply render the latest state available in the store whenever the browser is ready to paint a new frame. This decoupling ensures that the simulation's logical progression is stable and unaffected by rendering performance. It also enables advanced visual techniques, such as interpolating between the previous state and the current state during the render cycle to create exceptionally smooth animations, without compromising the integrity of the underlying simulation. This is a professional game development pattern that the proposed architecture enables by default.

#### Works cited

1. Marketing Mix Modeling: Everything You Need to Know - Analytic Edge, accessed October 13, 2025, <https://analytic-edge.com/blog/what-is-marketing-mix-modeling/>
2. Marketing mix modeling - Wikipedia, accessed October 13, 2025, <https://en.wikipedia.org/wiki/Marketing_mix_modeling>
3. What is MMM? Marketing mix modeling explained (with video) - Funnel.io, accessed October 13, 2025, <https://funnel.io/blog/what-is-marketing-mix-modeling-mmm-explained>
4. Media Mix Modeling (MMM): The Complete Guide for 2025 - Invoca, accessed October 13, 2025, <https://www.invoca.com/blog/media-mix-modeling>
5. Marketing Mix Modeling: A Complete Guide for Strategic Marketers | Measured®, accessed October 13, 2025, <https://www.measured.com/faq/marketing-mix-modeling-2025-complete-guide-for-strategic-marketers/>
6. Pure Functions with Typescript - Oscar - Medium, accessed October 13, 2025, <https://oscar-reyes.medium.com/pure-functions-with-typescript-651c3d2167d8>
7. You Need to Know About Pure Functions & Impure Functions in JavaScript, accessed October 13, 2025, <https://dev.to/codeofrelevancy/you-need-to-know-about-pure-functions-impure-functions-in-javascript-57>
8. Pure Functions in JS: Avoiding Unwanted Problems | by Fernando Doglio | Bits and Pieces, accessed October 13, 2025, <https://blog.bitsrc.io/pure-functions-in-js-avoiding-unwanted-problems-a67974f34aa3>
9. JavaScript to TypeScript Migration: 5 Best Practices | Blazemeter, accessed October 13, 2025, <https://www.blazemeter.com/blog/javascript-to-typescript>
10. Migrating from JavaScript to TypeScript: A Step-by-Step Guide for Success - Medium, accessed October 13, 2025, <https://medium.com/@schaman762/migrating-from-javascript-to-typescript-a-step-by-step-guide-for-success-c8fce2d8b0b6>
11. Mastering Code Refactoring with ReactJS and TypeScript | TO THE NEW Blog, accessed October 13, 2025, <https://www.tothenew.com/blog/mastering-code-refactoring-with-reactjs-and-typescript/>
12. “Adstock” and the Long and Short of Advertising - Impression Digital, accessed October 13, 2025, <https://www.impressiondigital.com/blog/adstock-effect/>
13. Advertising adstock - Wikipedia, accessed October 13, 2025, <https://en.wikipedia.org/wiki/Advertising_adstock>
14. What is Adstock Effect in Marketing? - Sellforte, accessed October 13, 2025, <https://sellforte.com/blog/what-is-adstock-effect-in-marketing>
15. Media saturation and lagging | Meridian - Google for Developers, accessed October 13, 2025, <https://developers.google.com/meridian/docs/advanced-modeling/media-saturation-lagging>
16. MMM: Saturation Curves - Indaru, accessed October 13, 2025, <https://www.indaru.com/mmm-saturation-curves/>
17. Hill Function and Its Implementation: -A Strategic Overview | by Rajiv ..., accessed October 13, 2025, <https://medium.com/@mail2rajivgopinath/hill-function-and-its-implementation-a-strategic-overview-2ffd443f92a0>
18. Synergy in Business: Definition & Examples - Lesson - Study.com, accessed October 13, 2025, <https://study.com/academy/lesson/synergy-in-business-definition-examples-quiz.html>
19. Marketing Synergy - Monash Business School, accessed October 13, 2025, <https://www.monash.edu/business/marketing/marketing-dictionary/m/marketing-synergy>
20. What is Synergy? Types of Synergy, Corporate Synergy and Multi Product Synergy - YouTube, accessed October 13, 2025, <https://www.youtube.com/watch?v=IccE0WVOJwo>
21. Types of Interaction Effects in Market Mix Modeling (MMM) | by ..., accessed October 13, 2025, <https://medium.com/data-science/types-of-interaction-effects-in-market-mix-modeling-mmm-95247f3de36e>
22. Free Download Synergy Identification Matrix Template - Meegle, accessed October 13, 2025, <https://www.meegle.com/en_us/advanced-templates/mergers_and_acquisitions/synergy_identification_matrix_template>
23. How to Test TypeScript with Jest? - GeeksforGeeks, accessed October 13, 2025, <https://www.geeksforgeeks.org/typescript/how-to-test-typescript-with-jest/>
24. oop - TypeScript functions within functions vs. private functions ..., accessed October 13, 2025, <https://stackoverflow.com/questions/77824023/typescript-functions-within-functions-vs-private-functions>
25. import of single functions from modules? good or bad? : r/javascript - Reddit, accessed October 13, 2025, <https://www.reddit.com/r/javascript/comments/40n0yq/import_of_single_functions_from_modules_good_or/>
26. Documentation - Modules - TypeScript, accessed October 13, 2025, <https://www.typescriptlang.org/docs/handbook/2/modules.html>
27. What's the difference between internal and external modules in TypeScript? - Stack Overflow, accessed October 13, 2025, <https://stackoverflow.com/questions/12841557/whats-the-difference-between-internal-and-external-modules-in-typescript>
28. pmndrs/zustand: Bear necessities for state management in React - GitHub, accessed October 13, 2025, <https://github.com/pmndrs/zustand>
29. Large-Scale React (Zustand) & Nest.js Project Structure and Best ..., accessed October 13, 2025, <https://medium.com/@itsspss/large-scale-react-zustand-nest-js-project-structure-and-best-practices-93397fb473f4>
30. React State Management — using Zustand | by Chikku George | Globant - Medium, accessed October 13, 2025, <https://medium.com/globant/react-state-management-b0c81e0cbbf3>
31. Unit Testing vs. Integration Testing: 5 Key Differences and Why You Need Both | Codefresh, accessed October 13, 2025, <https://codefresh.io/learn/unit-testing/unit-testing-vs-integration-testing-5-key-differences-and-why-you-need-both/>
32. Jest, Typescript, and unit testing | by Simon Lutterbie | Medium, accessed October 13, 2025, <https://simon-lutterbie.medium.com/jest-typescript-and-unit-testing-27c6ba78f020>
33. Integration Tests with Typescript | by Conrad Mugabe - Medium, accessed October 13, 2025, <https://conradmugabe.medium.com/integration-tests-with-typescript-396b0162d0de>
34. How to Test a TypeScript App with Jest | A Step-by-Step Tutorial - Meticulous, accessed October 13, 2025, <https://www.meticulous.ai/blog/jest-typescript>
35. BUDGET - SIMULATION ACTIVITY - Rogue Credit Union, accessed October 13, 2025, <https://www.roguecu.org/media/jzhjyz0f/budget-simulation-amended.pdf>
36. Budget Game Projects Well - Jeff Tidball, accessed October 13, 2025, <https://www.jefftidball.com/2016/04/08/budget-game-projects-well/>
37. Budget Allocation Simulation: Dynamic Budget Allocation Models: A ..., accessed October 13, 2025, <https://fastercapital.com/content/Budget-Allocation-Simulation--Dynamic-Budget-Allocation-Models--A-Simulation-Perspective.html>
38. Marketing Simulations - Marketplace Business Simulations, accessed October 13, 2025, <https://www.marketplace-simulation.com/marketing-simulations/>
39. NewShoes - Marketing - Interpretive Simulations, accessed October 13, 2025, <https://www.interpretive.com/business-simulations/introduction-to-marketing-simulation/>
40. Browse thousands of Simulation UI images for design inspiration ..., accessed October 13, 2025, <https://dribbble.com/search/simulation-ui>
41. Oracle Analytics Cloud Simulator, accessed October 13, 2025, <https://www.oracle.com/webfolder/assets/data-visualization/index.html>
42. Digital Marketing simulation game | Learn from experience - Hubro Education, accessed October 13, 2025, <https://hubrosimulations.com/hubro-digital-marketing-simulation>
43. Digital Marketing Simulation for Education | GoVenture.net, accessed October 13, 2025, <https://www.goventure.net/dm>
44. Build the Game of Life with React and TypeScript - DEV Community, accessed October 13, 2025, <https://dev.to/toluagboola/build-the-game-of-life-with-react-and-typescript-5e0d>
45. Farcebook - a free and open-source incremental game with React+Typescript-based template : r/incremental\_gamedev - Reddit, accessed October 13, 2025, <https://www.reddit.com/r/incremental_gamedev/comments/13esxhy/farcebook_a_free_and_opensource_incremental_game/>

he "Go Ham" Prompt for Courson

Your Persona: Act as a world-class principal software engineer, UI/UX architect, and data visualization expert. Your mission is to transform a dense, highly technical academic document into a stunning, interactive, and educational single-page web application. The final product must be so intuitive and well-crafted that it feels like a premium, bespoke piece of software. This is not a simple report visualization; it is an interactive learning environment. Go ham on the details and build something unbelievable.

The Core Task:

Analyze the provided source document, "The Mathematical Foundations of a Coefficient Simulator," and create a single-file interactive web application that allows users to explore, manipulate, and intuitively understand its core mathematical concepts. The application must be a masterpiece of design, usability, and technical execution.

1. Deconstruct and Redesign the Information Architecture:

Do not simply replicate the document's linear structure. Your first task is to architect a new, superior user flow. I recommend a "Concept Sandbox" model with three main interactive modules that the user can explore:

Module 1: The MMM Engine Room: Focus on the core MMM mathematics (Adstock, Hill Curve, Synergy).

Module 2: The Attribution Lab: Focus on dissecting attribution and incrementality (Shapley, Causal Inference).

Module 3: The Optimization & AI Frontier: Focus on advanced concepts like ML, Optimization, and Game Theory.

Each module should be a self-contained, full-screen section that the user can navigate to. Within each module, every mathematical concept must be presented as an interactive widget or visualization.

2. Build Interactive "Concept Widgets" (The "Wow" Factor):

This is where you will demonstrate true genius. For each key mathematical concept, build a dedicated interactive component. The goal is to let users play with the parameters to build intuition.

Adstock & Hill Curve Widget:

Visualization: Create a single Chart.js line chart that shows "Marketing Response."

Interactivity:

Provide a slider labeled "Adstock Decay (λ)" from 0.1 to 0.9. As the user moves the slider, the shape of the response curve must animate in real-time to show how a longer/shorter ad decay affects impact over time.

Provide a slider labeled "Saturation (S)" and another for "Shape (n)." As the user adjusts these, the Hill Curve's S-shape must dynamically update, clearly visualizing the concept of diminishing returns.

Context: Display the Adstock and Hill Curve formulas next to the chart. As sliders are adjusted, dynamically update a "Key Insight" text block below the chart explaining what the user is seeing (e.g., "Notice how a high decay rate (λ) creates a long 'tail' of ad effectiveness.").

Bayesian Modeling Widget:

Visualization: Use Chart.js to plot two distribution curves on the same chart: a "Prior Belief" (e.g., a wide, uncertain bell curve) and a "Posterior Belief" (e.g., a narrower, more confident curve).

Interactivity: Include a button labeled "Introduce New Data." Each time the user clicks it, the "Posterior Belief" curve should animate, becoming narrower and shifting its peak, visually demonstrating how the model "learns" and becomes more certain.

Shapley Value Widget:

Visualization: Create a simple bar chart showing 3 marketing channels (e.g., "Social," "Search," "Email").

Interactivity: Provide checkboxes for each channel. As the user checks/unchecks channels, the bar chart must dynamically re-calculate and animate the "fairly attributed" conversions for the selected channels, visually representing the Shapley formula's logic of evaluating different channel coalitions.

3. Technical & Design Execution (Unbelievably Well Built):

Single File Mandate: All HTML, CSS (via Tailwind), and JavaScript must be in a single .html file.

Aesthetics:

Palette: Use a "Technical Blueprint" theme. Background: slate-50 (#f8fafc). Main Text: slate-800 (#1e293b). Primary Accent (for interactive elements, highlights): sky-500 (#0ea5e9). Formula Backgrounds: slate-100 (#f1f5f9).

Typography: Use 'Inter' for all text and 'Fira Code' for all mathematical formulas to ensure clarity and a modern, technical feel.

Layout: Use generous whitespace, clean lines, and a max-width container to ensure readability. The layout must be fully responsive and flawless on all screen sizes. No horizontal scrollbars, ever.

Libraries: Use Tailwind CSS and Chart.js, loaded from CDNs. No SVG or Mermaid.js.

Code Quality: Your JavaScript must be clean, efficient, and exceptionally well-organized, using modern ES6+ syntax.

User Experience: Interactions must be smooth, with subtle animations and transitions on hover and click events. The application should feel alive and responsive to the user's every action.

4. Output Constraints:

Produce only a single HTML file.

Do not include any explanatory text outside of the file.

You must include the following comment blocks at the top of the HTML file, filling them in with your strategic plan:

<!-- Chosen Palette: [Name of selected palette] -->

<!-- Application Structure Plan: [Your detailed plan for the "Concept Sandbox" architecture and user flow.] -->

<!-- Visualization & Content Choices: [Your detailed plan for each interactive widget, justifying your choices.] -->

<!-- CONFIRMATION: NO SVG graphics used. NO Mermaid JS used. -->