ChatGPT

Arcade Platform Payout System Upgrade Plan

1. Configurable Payout Split Percentages

Current Setup: The Arcade platform's payment router defines how incoming payments are split among various pools (operations, daily rewards, weekly rewards, and treasury). In `ArcadePaymentsRouterV2.sol`, these splits are represented in basis points via a `SplitBps` struct. By default, the contract sets fixed split percentages for each payment **kind** during initialization. For example, the **KIND_CREDITS** payments are initially split as 7.00% to ops, 79.05% to the daily pool, 13.95% to the weekly pool, and 0% to treasury, totaling 100%. Likewise, **Pro membership mints/renewals** (KIND_PRO_MINT and KIND_PRO_RENEW) default to 25% ops, 0% daily, 50% weekly, 25% treasury. These defaults are essentially hardcoded at deployment time via `_setKindSplits(...)` calls in the constructor.

Modifiability: Currently, the router **allows** updates to these splits post-deployment. It exposes an owner-only function `setKindSplits(uint8 kind, SplitBps calldata splits) external onlyOwner` which internally calls `_setKindSplits` to update the stored split ratios. There is also an event `KindSplitsSet(kind, opsBps, dailyBps, weeklyBps, treasuryBps)` that is emitted whenever splits are changed【38†】. This means payout percentages are **not truly immutable** – an admin (owner) can adjust them. However, to align with best practices, we will reinforce and extend this configurability:

- **Admin Control via Multisig:** We will ensure the `onlyOwner` (admin) role is held by a secure multisig address. By transferring contract ownership to a Gnosis Safe (or similar), any critical change (like adjusting splits) requires multisig confirmation. This mitigates risk of a single key compromise and aligns with governance standards.

- **Adjustable Splits Implementation:** We will keep the existing `setKindSplits` function and possibly enhance it with additional safety checks. For instance, we'll require that the sum of the basis points equals 10,000 (100%) before applying an update, preventing configuration mistakes:

```
function setKindSplits(uint8 kind, SplitBps calldata newSplits) external
onlyOwner {
    require(
        uint256(newSplits.opsBps) + newSplits.dailyBps +
        newSplits.weeklyBps + newSplits.treasuryBps == 10000,
        "Invalid split: does not sum to 100%"
    );
    _setKindSplits(kind, newSplits);
    emit KindSplitsSet(kind, newSplits.opsBps, newSplits.dailyBps,
newSplits.weeklyBps, newSplits.treasuryBps);
}
```

The `SplitBps` struct is a convenient way to pass all four percentages. By using basis points (uint16 fields), we maintain precision and avoid floating-point issues. Emitting the `KindSplitsSet` event (as is already done internally) notifies off-chain services and the frontend of the change so they can sync the new ratios.

- **Event Emission for Frontend:** Each update to payout splits will emit an event (as shown above) with the new percentages. The frontend can listen for `KindSplitsSet` events to update the displayed configuration in real time, ensuring transparency for users about how payments are being allocated. This event was already defined in V2; our implementation makes sure to call it whenever an update occurs (if not already emitted internally). No changes are needed in other contracts because the Payment Router holds the logic for splits.

By making payout split percentages adjustable by an admin multisig and logged on-chain, the system gains flexibility to tweak the economic model (e.g. if we later decide to allocate more to weekly rewards versus daily) without requiring a contract redeploy. This approach prioritizes **upgradability through configuration**: critical parameters can evolve with governance approval, while preserving the integrity of the deployed contracts.

2. Introducing a `processPayment` Flow and Payout Scheduling

To handle incoming payments and coordinate reward distribution, we will implement a comprehensive `processPayment` function (or set of functions) in `ArcadePaymentsRouterV2.sol` that covers Web2/Web3 hybrid payments and schedules outgoing payouts.

Web2/Web3 Hybrid Payment Logic: Many Arcade transactions may originate off-chain (e.g. a user paying with a credit card or through a centralized service for in-game credits). To securely bridge off-chain payments to on-chain events, we use an **EIP-712 signed quote** mechanism. The backend (a trusted quote signer) generates a Quote containing details of the purchase – buyer address, item SKU, kind (credits purchase, pro membership, etc.), payment token, amount paid, equivalent USD value, number of credits (if applicable), membership tier, expiration time, and a nonce. This Quote is signed off-chain and then passed to the `processPayment` function on-chain by the user.

The `processPayment` implementation will:

- **Validate the Quote Signature:** Using EIP-712 domain separation (with domain name "GruesomeArcade PaymentsRouter" and the contract's address), the contract will recover the signer from the Quote and ensure it matches the authorized quote signer (set via an `setQuoteSigner` owner function). We have helper functions like `_verifyAndConsumeQuote` in place to do signature recovery and mark the nonce as used (preventing replay). Only a valid, unexpired quote can proceed.

- **Accept Payment:** Depending on `payToken` in the Quote, the contract either expects a native ETH value with the call or does an ERC-20 transfer. If `payToken == address(0)` (meaning the user is paying in native ETH), `processPayment` will be `payable` and require `msg.value == quote.amountIn`. If it's an ERC-20 token (e.g. a stablecoin or other token), the user must have approved the router, and we call `SafeTransferLib.safeTransferFrom(payToken, buyer,` `address(this), amountIn)` to pull the funds in. We will leverage the provided `MockERC20` for testing this flow, and the contract includes `setTokenAllowed(token, bool)` to whitelist acceptable payment tokens (ensuring, for example, only our test token or mainnet tokens like mUSD are accepted by the router).

- **Execute Business Logic:** Once payment is received and verified, the router will route the funds according to the payout splits and perform any additional actions for the given `kind` of purchase:

- *Credits Purchase:* The router emits a `CreditsPurchased` event (with details like buyer and credits amount) to record the issuance of off-chain credits. Off-chain systems listen to this and credit the user's arcade account with the purchased credits. (If in the future credits become an on-chain token, this function could mint those tokens to the user instead.)
- *Pro Membership Mint:* The router calls the `ArcadeProAvatarV2` contract's mint function to mint a Pro membership NFT to the buyer. (The router knows the `proAvatar` contract address via an `setProAvatar` setter). The Quote's `tier` field would be used to mint the correct membership level. An event `ProMinted` is emitted as well. Payment funds are split as per the Pro mint splits.
- *Pro Membership Renewal:* Similarly, the router calls the `ArcadeProAvatarV2` to record a renewal (could be an extension of NFT expiration or a new NFT depending on implementation) and emits `ProRenewed`.
- *Generic Payments:* For any other kinds or SKU not requiring special handling, we implement a fallback (a `payOnly` scenario) that just logs the payment and routes the funds, emitting a `PaymentExecuted` event.

Under the hood, all these flows use a common internal function (like `_executePayment`) to handle the token transfers according to the split percentages. This function calculates the split amounts for ops, daily, weekly, treasury based on the `SplitBps` for the given payment kind. It then transfers each portion to the respective recipient address: the **ops** share to the ops wallet, **daily** share to the daily rewards pool, **weekly** share to the weekly rewards pool, and **treasury** share to the treasury vault. These recipient addresses are configurable via `setRecipients(opsWallet, dailyPot, weeklyPot)` and `setTreasuryVault(vault)` functions (owner-only). For example, if a user purchases credits for 100 mUSD, the router would transfer 7 mUSD to the ops wallet, ~79.05 mUSD to the daily pot contract, ~13.95 mUSD to the weekly pot, and nothing to treasury in that case. All transfers use `SafeTransferLib` to handle ERC-20 safely and avoid reentrancy issues (since it uses low-level call and checks return). By routing funds immediately, the contract minimizes the time it holds tokens, reducing risk.

- **Secure Off-chain Quote Verification:** The use of EIP-712 signatures ensures that all critical payment details were agreed upon by our backend (preventing users from arbitrarily calling purchase functions with fake prices or amounts). For added security, the `expiresAt` field in the Quote prevents old quotes from being reused, and each Quote's `nonce` is tracked to avoid duplication. This design allows a Web2 payment (credit card, etc.) to be confirmed by the server and then honored on-chain in a trust-minimized way.

Manual vs. Scheduled Payouts: Once funds have been allocated to the **daily** and **weekly** pots, we need to distribute those to players (winners of competitions, etc.) on the appropriate schedule. The Arcade ecosystem handles this via the `ArcadeEpochVault.sol` (reward vault). Our upgrade will ensure both **manual triggers** and **automated scheduling** are supported for payouts:

- *Daily Payouts:* The `ArcadeEpochVault` contract holds the accumulated daily reward tokens (mUSD). Each day's distribution is prepared off-chain (the team calculates winners and their rewards) and then an authorized account calls `publishEpoch(ymd, merkleRoot, totalAmount, signature)` on the vault to publish the Merkle root of that day's reward distribution. The vault contract then allows each winner to `claim(ymd, index, account, amount, merkleProof)` their share of the daily pot. This is already a secure, gas-efficient way to do daily payouts. For **manual triggers**, an admin can simply run the off-chain script to compute the Merkle tree and call `publishEpoch` at the end of the day. For **automation**, we can integrate a cron job or Chainlink Automation that invokes a script to do this daily at a set time. The contract is designed to enforce one distribution per day (identified by the date code) and will emit `EpochPublished` and `Claimed` events for transparency.

- *Weekly Payouts:* Weekly competitions can be handled similarly. We might reuse the same `ArcadeEpochVault` by designating one day of the week (e.g., Sunday's date) as the "weekly epoch" that aggregates the week's rewards, or deploy a second vault for weekly rewards. In either case, the concept is the same: publish a Merkle root for the weekly winners when the week concludes. Our router already separates a weekly pot share and sends those funds to a configured `weeklyPot` address (likely the same EpochVault or a parallel vault). **Manual trigger** for weekly distribution would be an admin call at week's end to publish the weekly Merkle root. **Scheduled** automation can similarly be set for every Friday or Sunday midnight.

By supporting both manual and automated triggers, we ensure the team has flexibility. Initially, the team might execute payouts manually (to monitor and verify everything closely). As the platform grows, we can switch to reliable automation (using a secure server or a decentralized scheduler) for consistent user experience.

Code Integration: We will add the `processPayment` (or unify existing purchase functions under it) to encapsulate the above logic. Pseudocode for the unified function is as follows:

```
function processPayment(Quote calldata quote, bytes calldata signature)
external payable {
    require(!paused, "Payments paused");
    require(block.timestamp < quote.expiresAt, "Quote expired");
    _verifyAndConsumeQuote(quote, signature);  // EIP712 signature check and
nonce mark

    // Collect payment
    if (quote.payToken == address(0)) {
        require(msg.value == quote.amountIn, "Incorrect ETH amount");
    } else {
        require(msg.value == 0, "ETH not expected");
        require(allowedTokens[quote.payToken], "Token not allowed");
        SafeTransferLib.safeTransferFrom(quote.payToken, msg.sender,
address(this), quote.amountIn);
    }

    // Execute based on kind
    if (quote.kind == KIND_CREDITS) {
        // No on-chain asset to mint, just emit event
        emit CreditsPurchased(msg.sender, quote.sku, quote.credits,
quote.usdCents);
    } else if (quote.kind == KIND_PRO_MINT) {
        ArcadeProAvatarV2(proAvatar).mintPro(msg.sender, quote.tier);
        emit ProMinted(msg.sender, quote.tier, quote.usdCents);
    } else if (quote.kind == KIND_PRO_RENEW) {
        ArcadeProAvatarV2(proAvatar).renewPro(msg.sender, quote.tier);
        emit ProRenewed(msg.sender, quote.tier, quote.usdCents);
    } else {
        // KIND_GENERIC or others
        emit PaymentExecuted(msg.sender, quote.sku, quote.usdCents);
    }

    // Split and route funds
    SplitBps memory splits = kindSplits[quote.kind];
    uint256 opsAmount = (quote.amountIn * splits.opsBps) / 10000;
    uint256 dailyAmount = (quote.amountIn * splits.dailyBps) / 10000;
    uint256 weeklyAmount = (quote.amountIn * splits.weeklyBps) / 10000;
    uint256 treasuryAmount = quote.amountIn - opsAmount - dailyAmount -
weeklyAmount;
    // Transfer each portion
    if (opsAmount > 0) SafeTransferLib.safeTransfer(quote.payToken, opsWallet,
opsAmount);
    if (dailyAmount > 0) SafeTransferLib.safeTransfer(quote.payToken, dailyPot,
dailyAmount);
    if (weeklyAmount > 0) SafeTransferLib.safeTransfer(quote.payToken,
weeklyPot, weeklyAmount);
    if (treasuryAmount > 0) SafeTransferLib.safeTransfer(quote.payToken,
treasuryVault, treasuryAmount);
}
```

Note: The above is a simplified illustration. The actual contract code will handle some details differently (e.g., using stored `kindSplits`, and our router already has internal functions like `_executePayment`). Also, the Quote's `buyer` field could allow the platform to pay on behalf of someone else, but here we used `msg.sender` assuming the buyer is calling directly.

This implementation covers both the **Web2/Web3 payment integration** (via signed quotes and token transfers) and sets the stage for **scheduled payouts** (by segregating daily/weekly funds to dedicated contracts or addresses to be distributed at intervals). Using EIP-712 signing, a hybrid approach ensures off-chain processes (like fiat payments or complex reward computations) are anchored to on-chain events in a secure manner. We integrated a mock ERC20 (the provided `MockERC20.sol`) in tests to simulate different tokens and will ensure that switching to mainnet tokens is as easy as updating an address via the admin functions (`setTokenAllowed`, `setTreasuryVault`, etc.) once deployed.

3. Secure User Payouts and Event Logging

It's crucial that once funds are allocated to reward pools, they are paid out to users securely and transparently. The Arcade contracts already incorporate secure patterns for payouts; we will review and bolster these as needed:

- **Merkle-Proof Claims (Reward Pool):** The `ArcadeEpochVault` contract is designed for secure payouts via Merkle roots. After an epoch (daily or weekly), an authorized account publishes the Merkle root of the distribution along with the total amount to be paid【42†】. This ensures the contract knows exactly how much should be allocated and prevents malicious changes. Users then claim their portion by providing a proof that their address and amount are included in the published root. The vault contract verifies the proof and pays the user. Each claim is marked in a bitmap to prevent double-claims. The actual token transfer to the user is done via `SafeTransferLib.safeTransfer` (ensuring even non-standard ERC-20s are handled). An event `Claimed(account, amount, epochId)` is emitted on each successful claim【41†】, which serves as a public log of payout. We will double-check that the vault uses reentrancy guards or the Checks-Effects-Interactions pattern around claims (common practice to prevent reentrant attacks when transferring tokens to user-provided addresses). If not present, we will add a simple `nonReentrant` modifier (from OpenZeppelin's ReentrancyGuard) to the claim function to be safe.

- **Direct Payouts from Treasury:** In some cases, the team might need to send funds directly from the treasury (e.g., special rewards or migrations). The `ArcadeTreasuryVault.sol` allows the owner to withdraw ERC-20 tokens to a specified address. This function (`withdrawERC20(token, to, amount) external onlyOwner`) is restricted to the multisig (owner) and uses safe transfers as well. Each withdrawal will emit the standard ERC-20 `Transfer` event from the token contract, and we can also add a custom event like `TreasuryWithdrawal(token, to, amount)` for easier tracking if desired. Since this is an admin-only function, the risk is low, but for transparency the multisig can signal or record the reason for each withdrawal off-chain as well. We recommend using the multisig's transaction notes or a community announcement whenever a manual treasury payout is done, in addition to on-chain logs.

- **Logging and Transparency:** All payout actions should be logged by events:

- When the router allocates funds, events like `PaymentExecuted`, `CreditsPurchased`, `ProMinted`, `ProRenewed` include info on who paid and how the funds were split (the `PaymentExecuted` event or a separate log can include the amounts to each pool for full transparency).
- When daily/weekly rewards are published, `EpochPublished` (with epoch id and total amount) is emitted.
- On each user claim, `Claimed(account, amount, epoch)` is emitted.
- On treasury withdrawal, a `Withdrawal` event (if we add it) or at least the token's `Transfer` event is emitted.

These on-chain records enable the community and auditors to trace all reward flows. For example, one can observe the total daily reward amount published each day and verify that it matches the sum of individual claim events for that day, ensuring no funds vanished. We will add any missing events that improve this traceability. In our review, the key events are already present. For instance, the router's `KindSplitsSet` and `TokenAllowed` events log configuration changes, and `PausedSet` logs emergency pauses, which is good for security monitoring.

- **Secure Fund Handling:** Both the router and vault use established secure coding practices. The router, after this upgrade, will not custody funds for longer than a single transaction: it immediately diverts payments to their respective vaults or addresses. The Epoch vault does custody the daily rewards until users claim, so we ensure its operations are simple (only hold and transfer out on claim) and protected (only authorized publish, and no complex logic that could be exploited). We will also ensure the contracts have appropriate ownership separation: for example, the PaymentsRouter's owner (governance multisig) can change config but **cannot drain user reward funds** directly – only the EpochVault handles user claims, and the multisig cannot arbitrarily take funds from the EpochVault because it lacks a generic withdraw (it can only publish new roots or update the oracle signer). This separation of concerns means even an admin cannot bypass the Merkle distribution mechanism to take daily reward funds, which is an important security feature.

In summary, the payout mechanisms in place (Merkle proofs for player rewards, multisig-controlled treasury vault for project funds) already address security and transparency. Our contributions are to double-check for any missing guards or events and add them where needed. After modifications, the system will securely handle direct user payouts through **claim proofs** or authorized withdrawals, with every transfer transparently logged on-chain.

4. Developer Action Plan (Terminal Workflow with VS)

To implement and deploy these upgrades, we'll follow a structured action plan. This plan is written as a sequence of steps that can be executed in a development terminal and through collaborative Git workflows. We assume we are working with developer **VS** and using tools like Hardhat, Foundry, Node.js, and Git. The target deployment environment is the Linea Sepolia testnet.

Step 1: Environment Setup
Ensure the development environment is ready:

```
# Navigate to the project directory
cd gruesome-arcade-contracts-v2.1

# Install dependencies if not already (Hardhat, ethers, etc.)
npm install

# If using Foundry for tests as well:
forge install
```

Both Hardhat and Foundry are available, so we can use Hardhat for deployment scripts and Foundry (forge) for high-speed solidity unit tests. Make sure you have a Linea Sepolia RPC endpoint configured in Hardhat

(`hardhat.config.js`) and proper private keys or a deployer account set up (likely through environment variables).

Step 2: Branch and Collaborate
Create a new git branch for these changes and push it so that VS can collaborate:

```
git checkout -b feature/payout-upgrades
git push origin feature/payout-upgrades
```

Communicate in our team channel that this branch is ready for collaboration. VS can pull the branch and start reviewing or adding UI changes in parallel while I work on the contracts.

Step 3: Implement Contract Changes
Open the Solidity contracts in your editor (VS Code or similar) and apply the modifications: - *Adjustable Splits:* In `ArcadePaymentsRouterV2.sol`, verify the `setKindSplits` function exists and modify it if necessary to include the sum validation and ensure it emits `KindSplitsSet` properly. The default split values set in the constructor should be moved into an initializer function if we convert to proxy, but since we use direct deployment, just double-check they sum correctly. No changes needed in `ArcadeEpochVault.sol` for this part, as it already just holds funds. - *Process Payment:* Still in `ArcadePaymentsRouterV2.sol`, implement the unified `processPayment` function (or refine existing functions `purchaseCredits`, `mintProAvatar`, etc.). Leverage the `_verifyAndConsumeQuote` for signature checks and `_executePayment` for fund routing. Essentially, consolidate the logic described in Section 2. Make sure to handle ETH vs ERC20 correctly (Hardhat's console can be used to test both scenarios). - *Manual/Scheduled Payout Hooks:* In `ArcadeEpochVault.sol`, add a `nonReentrant` guard to `claim` if not present (import `ReentrancyGuard`). Also, consider adding an owner-only `emergencyWithdraw` that could be used to move funds out in a worst-case scenario (and immediately pause the contract), but use caution – such a function can undermine the trust in the reward pool, so we might skip it or gate it behind a time lock if ever added. Since the requirement didn't explicitly ask for an emergency withdraw, it's probably best to avoid it to keep the vault immutable for users. Instead, ensure `publishEpoch` requires a valid signature from a designated oracle signer (or the multisig) – this was likely already implemented via an `OracleSignerUpdated` event and check on `publishEpoch`. Confirm that and adjust if needed (for example, if we want the multisig to directly call `publishEpoch`, we can also allow owner as a signer). - *Events and Logging:* Add any `emit EventName(...)` calls where needed. For example, if we create a new `TreasuryWithdrawal` event in `ArcadeTreasuryVault`, add `emit TreasuryWithdrawal(token, to, amount)` in the `withdrawERC20` function. Similarly, ensure `PaymentExecuted` (or a similar event) is emitted after funds are split in the router to log the actual amounts that went to each pool (this can help with transparency). - *Upgradability Consideration:* Since these contracts are not using proxy patterns (constructor is present), the upgrades are applied via redeploying new versions (ArcadePaymentsRouterV3, etc.). We'll note in the README that after testnet validation, the mainnet deployment should migrate state (if any) or simply replace addresses for new purchases while keeping old vaults for claims. For now, our testnet deployment will be fresh. (If we wanted to use a proxy, we'd refactor to `initialize()` functions and deploy proxies, but that may be overkill here given the timeline. We prioritize making as much configurable as possible to reduce the need for redeploys).

After editing, run our Solidity linters/formatters (e.g., `npm run lint:sol`) to ensure code style is consistent.

Step 4: Update Tests
Write or update **unit tests** to cover the new functionality: - Using Foundry (in `src/test` or similar), add tests for `setKindSplits` ensuring only owner can call and that it rejects invalid sums and accepts valid changes (and that the event is emitted). - Test the `processPayment` flow with a dummy quote: use Foundry to sign an EIP712 message (we can import the EIP712 libraries or simply call the internal `_hashTypedDataV4` with a known private key to simulate a signature). Verify that calling `processPayment` transfers funds correctly to each recipient (you can query balances of `opsWallet`, `dailyPot`, etc. after call). - In Hardhat (JavaScript/TypeScript tests), you might simulate a full flow: e.g., have a user "buy credits" by calling `processPayment` and then have them claim from `ArcadeEpochVault`. Hardhat can use ethers.js to sign the quote. Ensure the claim succeeds and the correct amount is received. - Test edge cases: expired quotes, reused nonces (should fail), unauthorized token, paused router, etc. Also test that non-owner cannot change splits or other admin settings. - If possible, test the schedule: e.g., call `publishEpoch` as the owner, then have multiple "users" claim. Verify events `EpochPublished` and `Claimed` are emitted and that double claiming is prevented.

Run the test suites:

```
# Run Foundry tests
forge test -vv

# Run Hardhat tests
npx hardhat test
```

Ensure all tests pass. Iterate on fixes if any fail.

Step 5: Deployment to Linea Sepolia
Once tests are green, prepare for deployment on Linea Sepolia: - Double-check the Hardhat config for the Linea Sepolia network (chain ID, RPC URL, and deployer key). - Update deployment scripts (e.g., `scripts/deploy.js` or a Foundry script) to deploy the modified contracts. We will deploy `ArcadePaymentsRouterV2` (or rename to V3 if we version bump), `ArcadeEpochVault`, `ArcadeTreasuryVault`, etc., in the correct order. For instance, deploy the vaults first, then the router (providing it with the addresses of vaults and other config like quote signer). We might also deploy `MockERC20` on Sepolia for testing if needed. - Execute the deployment:

```
npx hardhat run scripts/deploy.js --network lineaSepolia
```

This will output the deployed contract addresses.

- After deploying, call the initialization functions:
    - Use Hardhat tasks or a simple script to call `setRecipients(opsAddr, epochVaultAddr, epochVaultAddr)` – if using one vault for both daily and weekly – or provide a separate weekly vault if we deployed one. Also call `setTreasuryVault(treasuryVaultAddr)`.
    - Call `setQuoteSigner(<backend-signer-address>)` to authorize the off-chain signer.
    - If any tokens need to be allowed, call `setTokenAllowed(mockERC20, true)`.
    - These can be done in the deploy script or manually via Hardhat console/ethers.js.
- Verify the contract on Block Explorer (if applicable for Linea Sepolia) by submitting the source. This helps with transparency and debugging.

Step 6: Post-Deployment Testing
On the testnet deployment, perform a few scenario tests: - Have VS or another team member use the frontend (if available) or ethers scripts to simulate a purchase on Linea Sepolia. For example, simulate a credit card payment by directly calling `processPayment` with a signed quote (we can use a throwaway private key as the quote signer for this test). - Publish a fake daily epoch and attempt a claim to ensure the vault distribution works end-to-end with the actual deployed contracts. - Monitor the events in the block explorer or Hardhat console to confirm everything is logging as expected.

Step 7: Code Review & Merge
Open a Pull Request on GitHub for the `feature/payout-upgrades` branch. Both myself and VS (and any other reviewers) will go through the diff. Key things to scrutinize: security of the new functions (no unchecked `transfer` of large amounts without limits, proper `onlyOwner` on admin funcs, etc.), correctness of math for splits, and that no storage slot conflicts or uninitialized variables exist. Once everyone is satisfied, merge the PR into the main branch.

Step 8: Frontend Coordination
Coordinate with VS on integrating the changes with the frontend (admin dashboard). Ensure the ABI changes (new events, new function `processPayment`) are reflected in the front-end code. Likely, VS will handle the UI (addressed more in the next section), but we will support with any contract addresses or ABIs needed. We should also update the project’s README and documentation (e.g., `DEPLOYMENT_CLARIFICATIONS.md`) to describe these new features, so that both developers and the community understand the new payout system controls.

By following this action plan in the terminal and through our dev tools, we cover the implementation, testing, and deployment of the payout split controls and payment processing features in a safe, collaborative manner.

5. Admin Dashboard Updates for Payout Controls and Scheduling

To complement the smart contract upgrades, the Arcade project's admin web interface (likely a Vercel-hosted app) needs new components for managing payout splits and scheduling payouts. Below are the tasks and prompts for the front-end (UI/UX) team to implement these features:

- **Payout Split Configuration UI:** In the admin panel, create a new section (e.g., "Payout Settings" or under an existing Settings page) that displays the current split percentages for each payment category and allows editing. This should include:
- A table or form showing current **Ops %, Daily %, Weekly %, Treasury %** for each `kind` of payment (Credits, Pro Mint, Pro Renew, etc.). Fetch these values from the `ArcadePaymentsRouterV2` contract via an on-chain call (or subgraph, if available). The contract has a public mapping or getter for `kindSplits` that can be used to populate these fields.
- Input fields (number inputs or sliders) to adjust the percentages. Consider using a slider or spinner that ensures the total equals 100%. Alternatively, allow editing three fields and calculate the fourth automatically to sum to 100%. Provide real-time feedback if the sum is off.
- A "Save" or "Update" button. When clicked, the UI should:
    1. Validate inputs (sum to 100, each value within 0-100%).
    2. Prompt the admin to confirm the transaction (since this will trigger an on-chain tx).
    3. Call the `setKindSplits(kind, {opsBps, dailyBps, weeklyBps, treasuryBps})` function via web3/ethers. Ensure the wallet used is the multisig or a delegated executor for it. (If the multisig cannot directly connect to UI, consider using Gnosis Safe's transaction service or a module that allows proposing the change through the UI).
    4. On success, display a notification and update the displayed values. The UI can also listen for the `KindSplitsSet` event to reflect updates trustlessly.

- **UX considerations:** Clearly label what each pool means (e.g., tooltip: "Ops = operational costs", "Daily = Daily rewards pool", etc.) so the admin is confident in what they're changing. Also show the effective basis points or just use percentage for simplicity (we can input in whole percent and convert to bps under the hood).

- **Manual Payout Trigger Interface:** Provide a tool for admins to manually initiate payouts if needed:

- For **daily rewards**: Display the last epoch (date) published and perhaps the total distributed (this can be read from an event or stored value in the contract). Also display whether an epoch for today has been published or not. If not, allow the admin to trigger it. This could be a button "Publish Today's Rewards" which when clicked, asks for the Merkle root file upload or a link. In practice, the admin will have a JSON or CSV of winners that a script turns into a Merkle root and signature. The UI should allow the admin to input:
    - The date (default to today's date).
    - The Merkle root (maybe as a hex string).
    - The total reward amount for that day (for reference/display).
    - Then call the `publishEpoch(ymd, root, total, signature)` via web3. The signature might be generated server-side by an oracle service; if so, the UI might need to fetch it from an API or have the admin paste it.
- For **weekly rewards**: Similar approach, but perhaps with a dropdown to select which week or an auto-detection that "the week ending on YYYY-MM-DD" is ready to publish. If we use the same

contract for weekly, use the corresponding id (perhaps we use Sunday's date as epoch id). If a separate contract, that would have its own publish function.

- Include a confirmation step, as these actions cannot be easily undone on-chain. Once published, inform the admin and display the new epoch's info. The UI should update to show that the current epoch is now published and maybe list the top winners (if we want to surface that data by reading from IPFS or the backend).

- **Automated Schedule Indicator:** If the project decides to automate payouts via a backend cron or Chainlink, reflect that in the UI:

- For example, show a toggle or note "Daily payouts: **Automated at 00:00 UTC** via backend" if automation is on. If the admin can toggle automation, provide a switch. (Toggling might just be an off-chain setting; unless we have an on-chain flag to allow automatic publishing by a certain account. In our contracts, there isn't an on-chain toggle – the automation would simply call publishEpoch on schedule. So this UI element might just be informational.)
- Show a countdown or scheduled time for the next payout. This can be a simple calculation: if it's before midnight, "Next payout scheduled in X hours". This helps the admin (and possibly users via an admin dashboard view) know when to expect distribution.

- If automation fails or is turned off, the UI should alert the admin that a manual action is required ("No daily distribution has been published for today yet!" after the expected time).

- **Admin Payout Overview:** Create an overview page showing key metrics for transparency:

- Total funds currently in Daily pool and Weekly pool (query the `balanceOf(mUSD)` for the vault address or track within the contract if available).
- Pending rewards vs distributed (maybe how much was allocated this epoch and how much claimed so far – the vault could provide `totalAmount` for an epoch and we can sum claim events).

- Recent payout events log: a list of recent `EpochPublished` and large `Claimed` events, or at least an indicator that "Daily rewards for 2025-01-05 published: 5,000 mUSD distributed to 120 players" for example. This not only helps admins but is also great for community transparency if this admin panel has sections visible to moderators or read-only users.

- **Front-end Integration of `processPayment`:** Although primarily an internal function, ensure the front-end uses the correct function for processing payments. If previously the UI called separate endpoints or used multiple functions for different purchase flows, it can now simplify to calling `processPayment` with the signed quote from the backend. Coordinate with the backend team such that when a user completes a Web2 payment, the backend provides the UI with an EIP-712 signature and Quote payload. The UI (if it's non-custodial) will then prompt the user's wallet to call `processPayment(quote, sig)`. This will improve user experience by consolidating all purchases into a single transaction flow. Make sure to handle the case of paying with ETH vs ERC20 in the front-end: if the quote says `payToken = 0x0` (ETH), the web3 call must include `value: amountIn` in the transaction request. If an ERC20, ensure the user has approved the router for that `amountIn` beforehand (the UI should detect if allowance is needed and guide the user to approve the token first).

- **Testing the UI Changes:** Once implemented, test the following scenarios in the staging environment:

- Adjust payout splits to new values, ensure the transaction succeeds and the new percentages persist (you can cross-verify via reading the contract).
- Try inputting an invalid set of percentages (like not summing to 100) – the UI should prevent submission or show an error.
- Use the manual publish tools: simulate publishing a daily rewards root (perhaps use a dummy Merkle root for testing). Ensure the transaction is sent and events update. Then try to claim from a user account to ensure the published root is effective (this might require a custom script or an advanced UI not in admin panel).
- Confirm that the `processPayment` integration works end-to-end for each kind: e.g. simulate a credit purchase through the UI flows (perhaps with a test button that calls backend for a quote and then calls the contract). For pro mint, since it will actually mint an NFT, ensure the NFT shows up in the user's wallet after.
- Check that all new UI elements are responsive and don't break the layout, and that sensitive actions are properly gated (only show admin controls to addresses with admin rights, etc., perhaps by checking the connected address against an allowed list or by seeing if they can call an owner function).

These admin UI enhancements will empower the team to manage the Arcade economy in real-time and give clear visibility into the reward system's operation. By making these controls accessible and user-friendly, we reduce the chance of misconfiguration and build trust with the community through transparency (since admins can easily relay this information or even share read-only dashboards). The end result is a more **configurable**, **observable**, and **controllable** reward system accessible through the Arcade platform's interface.

6. Competitive Payout System Features & Best Practices

In designing Arcade's payout/reward system, it's useful to consider what features are most desired in comparable Web3 gaming and reward platforms. Many successful platforms have converged on a set of best practices that increase user engagement, fairness, and sustainability. Below is a summary of key features and how they apply to Arcade:

- **Frequent and Tiered Reward Cycles:** Top platforms engage users with frequent rewards cycles (daily quests, weekly tournaments, seasonal or monthly competitions). Daily and weekly rewards keep short-term engagement high, while larger seasonal rewards maintain long-term interest [1]. Arcade follows this model with daily and weekly pots. The best practice is to ensure these cycles are consistent (rewards drop on schedule) and tiered appropriately — daily rewards for regular activity and weekly for competitive outcomes, for example. This creates a layered incentive structure where casual players get frequent small wins and hardcore players strive for the bigger weekly prize.

- **Configurability & Dynamic Tokenomics:** The ability to adjust reward parameters is crucial as the user base and token economy evolve. Successful projects implement **dynamic token pools** that scale with growth or economic conditions [2]. For instance, if the user base doubles or token value fluctuates, the reward pool can be adjusted to remain attractive but not inflationary. Arcade's approach to make split percentages and total distributions adjustable by admins (with multisig and community oversight) fits this best practice. We've ensured that via contract controls, changes can be made to the distribution formula (e.g., allocating more to treasury if needed or adjusting how much goes to daily vs weekly) without a code redeploy. This configurability should be coupled with clear communication — any change in reward allocations should be transparent to users to maintain trust.

- **Transparency and Verifiability:** Users in Web3 gravitate toward systems where they can verify outcomes. Platforms often publish the details of reward calculations, either on-chain or off-chain with cryptographic proofs. For example, using Merkle roots for distributions (as Arcade does) is a common approach to ensure anyone can verify that the list of winners and amounts was not tampered with. Another aspect of transparency is real-time visibility: events emitted on-chain (which we have for every important action) allow community members or independent auditors to track the flow of funds. Some projects also provide a public dashboard showing how rewards are accumulating and distributed, which we plan to emulate via our frontend. In summary, all moving parts of the reward system should be auditable – from the initial payment splits (which are on-chain variables) to the final claim by a user (verified by an event and token transfer).

- **Multi-Token and Stablecoin Support:** Competitive reward systems often use one primary reward token (sometimes the game's native token or a stablecoin to provide a stable reward value). The trend, however, is to be flexible with **which tokens can fund rewards or be given out**. For example, a platform might accept various cryptocurrencies as payment (ETH, stablecoins, etc.) but internally convert them to a stable unit for distributing to winners, to avoid volatility. Arcade is already aligned with this by using mUSD (a stablecoin) as the reward currency, which is great for maintaining consistent reward value. The contract design also allows accepting other tokens for payment (we have a token allow list) and funnels non-stable payments to treasury if needed. Many platforms choose stablecoins for rewards to protect players from market swings, or use their native token if they want to give a stake in the ecosystem. Supporting multiple payment tokens can broaden the user base (more ways to pay), but it's important to have a clear, single denomination for rewards to keep things fair. Arcade's design of converting everything to a stable reward pool each epoch is in line with industry practice.

- **Hybrid Off-chain/On-chain Architecture:** Nearly all high-throughput Web3 dApps (especially games) use a hybrid approach for scalability and better UX. Heavy computations or frequent interactions (like calculating leaderboard rankings, handling micro-transactions, etc.) are done off-chain, while the results are anchored on-chain for security. We see this in projects like **DAR's Quest System**, which calculates complex quest outcomes off-chain but then does a monthly on-chain airdrop of tokens to winners [1] [2]. Arcade similarly uses off-chain logic for quote signing (fiat payments and price calculations) and for computing reward distribution (the Merkle trees), and uses on-chain transactions to finalize those results. This approach provides **the best of both worlds**: it's efficient and can handle large numbers of players without prohibitive gas costs, but it still gives players cryptographic proof that the rules were followed. The EIP-712 signatures and Merkle proofs are key components of this hybrid model, ensuring trustlessness where it matters (funds distribution) while leveraging off-chain speed for everything else. Users generally prefer this because it means fewer transactions on their end (one claim to get a week's worth of rewards rather than multiple small claims) and the platform can iterate on off-chain algorithms rapidly.

- **Reward Frequency vs. Gas Costs:** One challenge is balancing how often users receive rewards with the gas cost of claiming those rewards. Some platforms choose to accumulate smaller daily rewards into a weekly claim to save users transaction fees (e.g., **monthly airdrops** instead of daily claims [1]). Arcade's daily vs weekly structure might consider allowing players to accumulate daily rewards and claim once a week if gas is a concern on Ethereum mainnet; on Linea, costs are lower, but it's still good to be mindful. The system we built can accommodate that by simply not forcing a claim every day – users could wait and claim multiple days at once if the frontend supports it (because each day is a separate Merkle tree, they'd have to do one per day though, unless we also implement a weekly roll-up of daily rewards). An improvement could be to introduce a "rolling up" mechanism for unclaimed rewards after a certain period, but this adds complexity. For now, our approach is to publish daily but it could be adjusted to weekly only if desired to reduce claim frequency. The key is we have **flexibility** to adjust frequency via the contract (we could choose to only use weekly and set dailyBps to 0 if we found daily distribution too costly or unnecessary, for instance).

- **Community Governance and Fairness:** In truly decentralized projects, the community often has a say in reward parameters (through governance votes). While Arcade is currently admin-controlled, it's good to note this trend: platforms like **Compound or Uniswap** let token holders vote on changes to protocol parameters. In a gaming context, that could mean players voting on how much of the revenue goes into prize pools versus treasury. Our implementation with multisig control is a stepping stone – it can later be handed over to a DAO or governance module if Arcade issues a token. Fairness perceptions also matter: by using algorithms and formulas that are public (or even open-source for the off-chain parts) and by not giving anyone (not even admins) unilateral power to siphon rewards, we ensure the system is viewed as fair. This encourages player trust and long-term participation.

In conclusion, the upgraded Arcade payout system aligns well with industry-best features: it has frequent reward intervals, the flexibility to tweak and sustain the economy, transparency through on-chain events and cryptographic proofs, support for stablecoin-based rewards and various payment tokens, and a robust hybrid design. These features collectively create a competitive advantage, as they address both developer needs (configurability, upgradability) and user desires (consistent rewards, trust, and ease of use). By learning from other Web3 platforms and implementing these best practices, Arcade is well-positioned to deliver a rewarding and trustworthy experience to its community. [1] [2]

---

[1] [2] DAR's Web3 Quest System and the Future of Play-to-Earn Gaming | Bitget News
https://www.bitget.com/news/detail/12560604937970