Technical Specification and Implementation Report: "Get Rich Fast" – A Serverless Daily Financial Simulator

1. Architectural Overview and Design Philosophy

The development of "Get Rich Fast," a daily financial simulator operating within the Reddit Devvit ecosystem, represents a convergence of serverless architecture, generative artificial intelligence, and community-driven gaming dynamics. This report serves as an exhaustive technical manual and implementation guide, detailing every facet of the application's construction from the configuration of the runtime environment to the pixel-perfect rendering of the "Seinen/Noir Manga" user interface.

The core objective is to simulate the volatile, high-stakes atmosphere of day trading, heavily influenced by the cultural phenomenon of retail trading communities like r/WallStreetBets.[1] The application distinguishes itself through a rigorous adherence to a specific aesthetic—high-contrast, black-and-white noir visual language—and a technical architecture that runs entirely serverless, relying on the Reddit Devvit runtime for computation, scheduling, and storage.

1.1 The Serverless Paradigm on Devvit

The Devvit platform imposes a unique set of constraints and capabilities that fundamentally shape the application architecture. Unlike traditional containerized applications where a persistent server maintains state in memory, the Devvit environment is ephemeral. Functions, triggers, and scheduled jobs spin up on demand and terminate immediately upon completion. This necessitates a stateless architectural pattern where all persistence is offloaded to Redis, and all logic is event-driven.[3]

For "Get Rich Fast," this means there is no "game loop" running in the background in the traditional sense. Instead, the game moves forward in discrete ticks triggered by the Devvit Scheduler. The state of the market does not exist continuously in memory; it is re-hydrated from the database (Redis) every time a user loads the post or the daily scheduler fires.[5] This report outlines a system design that embraces these constraints, utilizing the scheduler capability to create the illusion of a living, breathing market that updates reliably every 24 hours.

1.2 Aesthetic and Tonal Integration

The visual and narrative direction is strictly defined by a "Seinen/Noir Manga" aesthetic.[7] This choice dictates a high-contrast user interface, jagged geometric layouts, and a distinct lack of grayscale gradients in favor of stark black-and-white compositions. The color palette is

intentionally restricted to black (#000000), white (#FFFFFF), and two signal colors representing market sentiment: "Bullish Green" and "Bearish Red."
This visual language serves a functional purpose beyond mere style. The "jagged" borders and stark contrasts mimic the cognitive load and stress of high-frequency trading. The interface is divided into three distinct panels—News, Advisor, and Action—creating a Z-pattern layout that guides the user's eye from the daily context (News) to the character-driven advice (Advisor) and finally to the decision point (Action).[8]

1.3 Data Flow and System Architecture

The application relies on a unidirectional data flow initiated by a time-based trigger.

1. **Trigger:** The Devvit Scheduler fires a daily_market_open event at a configured time (e.g., 13:00 UTC).
2. **Generation:** The application server receives this event and constructs a prompt for the Google Gemini API.
3. **External Computation:** The Gemini 1.5 Flash model generates a structured JSON payload containing the day's market headline, narrative subtext, character dialogue, and market movement percentage.
4. **Persistence:** The server validates this payload and persists it to Reddit's internal Redis storage, archiving the previous day's state.
5. **Consumption:** When a user visits the subreddit and loads the game post, the Webview component mounts, fetches the current state from Redis via the server, and renders the interface.

The rejection of complex architecture diagrams in this report requires a detailed textual breakdown of these interactions. The Scheduler acts as the heartbeat, the Server as the brain, Gemini as the creative engine, and Redis as the memory. The interactions between these components are strictly asynchronous, necessitating robust error handling and fallback mechanisms which will be detailed in the subsequent sections.

---

2. Platform Configuration: devvit.json

The devvit.json file is the nucleus of the application. It acts as the manifest that declares the application's identity, its entry points, and most critically, the permission scopes it requires to operate. In the security-conscious environment of Reddit's developer platform, capabilities such as making external HTTP requests or scheduling background jobs are disabled by default. We must explicitly opt-in to these features.

2.1 Permission Analysis and Security Scopes

To achieve the functional requirements of "Get Rich Fast," the configuration must enable three specific high-privilege scopes.

2.1.1 HTTP Egress (http)

The application relies on the Google Gemini API for its content generation. This requires the server to open an outbound connection to generativelanguage.googleapis.com. Devvit enforces a strict allow-list policy for HTTP requests to prevent abuse. While generativelanguage.googleapis.com is on the global allow-list of approved domains, the application manifest must still explicitly declare the intent to use the http capability and list the domains it will contact.[10] Failure to declare this will result in immediate runtime errors when the fetch command is invoked.

2.1.2 Scheduled Jobs (scheduler)

The "Daily" aspect of the simulator is non-negotiable. We cannot rely on user activity to trigger the new day's market update, as this would lead to inconsistent timing or race conditions where multiple users trigger the update simultaneously. The scheduler permission allows the registration of cron-like jobs that run independently of user interaction. This capability is the engine that drives the game's daily retention loop.[5]

2.1.3 Data Persistence (redis)

Redis is the only persistence mechanism available to Devvit apps for this use case. It is essential for storing the "Daily Scenario" (the shared state for all users) and the individual user portfolios (private state). Without the redis permission, the application would be strictly ephemeral, resetting completely on every page load.[6]

2.2 The Configuration Artifact

The following JSON structure represents the validated configuration for the application. It maps the daily_market_open task to a specific cron schedule and points the custom post type to the Webview entry point.

JSON

```json
{
  "$schema": "https://developers.reddit.com/schema/config-file.v1.json",
  "name": "get-rich-fast-sim",
  "version": "0.1.0",
  "description": "A noir-manga styled daily financial simulator powered by Gemini.",
  "main": "src/main.ts",
  "webroot": "src/webview",
  "permissions": {
    "http": {
      "domains": [
        "generativelanguage.googleapis.com"
      ]
    },
    "scheduler": true,
    "redis": true,
    "media": true
  },
  "scheduler": {
    "tasks": {
      "daily_market_open": {
        "cron": "0 13 * * *",
        "description": "Generates the daily market scenario at 1 PM UTC (Market Open)"
      }
    }
  },
  "custom_post_type": {
    "name": "DailySimulation",
    "entry_point": "src/webview/App.tsx"
  }
}
```

2.3 Scheduler Configuration Strategy

The cron expression 0 13 * * * is a deliberate design choice derived from financial market analysis. The New York Stock Exchange (NYSE) opens at 9:30 AM Eastern Time. Depending on Daylight Saving Time, this corresponds to either 13:30 or 14:30 UTC. By setting the trigger to **13:00 UTC**, we ensure the daily content is generated and available *just before* or as the US markets open. This synchronization maximizes the relevance of the content for the target audience, who are likely active during these hours.[1]
The tasks object in devvit.json defines the interface between the platform's scheduler service and our application code. The key daily_market_open serves as the event name that our server code will listen for. This decoupling allows us to change the schedule in the JSON file without modifying the TypeScript logic, providing operational flexibility.

---

3. Server-Side Architecture: src/main.ts

The src/main.ts file is the application's entry point. In the Devvit framework, this file is responsible for registering the application's capabilities with the runtime. It does not run continuously; rather, it defines *handlers* that the platform invokes in response to specific events (e.g., a scheduled job firing, a user clicking a menu item, or a post being rendered).

3.1 The Event-Driven Logic Flow

The server logic must orchestrate three distinct operational modes:

1. **The Scheduled Update:** Handling the background generation of content.

2. **The Interactive Render:** Serving the game UI to a user.
3. **The Administrative Override:** Allowing manual triggers for testing or correction.

The architecture uses a singleton pattern for the market state. Regardless of how many users are playing, there is only *one* "Market Scenario" for a given day. This shared state helps build a cohesive community experience where all players are reacting to the same "news" event, fostering discussion in the post comments.[11]

3.2 Implementation Details

The implementation leverages the @devvit/public-api SDK. A critical optimization in the render function is the use of useState with an asynchronous initializer. This pattern allows the server to fetch the necessary data from Redis *before* sending the initial HTML to the client. This "Server-Side Data Hydration" significantly improves the user experience by eliminating the "loading spinner" phase that typically occurs when a client-side app has to fetch data after mounting.[12]

3.3 Source Code Specification: src/main.ts

TypeScript

```typescript
import { Devvit, useState } from '@devvit/public-api';
import { generateDailyScenario } from './server/gemini.js';
import { backupDailyState, getLatestScenario } from './data/backups.js';

// Configure the Devvit instance with necessary plugins
Devvit.configure({
  http: true,
  redis: true,
  redditAPI: true,
  scheduler: true,
});

/**
 * SCHEDULER: Daily Market Generator
 * This handler is invoked by the Devvit platform based on the cron schedule
 * defined in devvit.json. It operates in a background context.
 *
 * Logic Flow:
 * 1. Acknowledge trigger.
 * 2. Call Gemini API to generate fresh content.
 * 3. Serialize and save the content to Redis.
 * 4. Log success/failure for developer visibility.
 */
Devvit.addSchedulerJob({
  name: 'daily_market_open',
  onRun: async (event, context) => {
    console.log('🔔 Market Bell Ringing! Generating daily scenario...');

    try {
      // 1. Generate Content via Gemini
      // This is an expensive async operation (1-3 seconds latency)
      const scenario = await generateDailyScenario(context);

      // 2. Persist Data
      // We use the date as the primary key for historical lookups
      const dateKey = new Date().toISOString().split('T');
      await backupDailyState(context, dateKey, scenario);

      console.log(`✅ Market Open for ${dateKey}: ${scenario.headline}`);

      // Note: In a production version, we might create a new Reddit Post here
      // using context.reddit.submitPost(), but for this simulator, we assume
      // the game lives in a pinned post that updates its internal state.

    } catch (error) {
      console.error('💥 Market Crash (Server Error):', error);
      // In a real deployment, we would trigger an alert or retry logic here
    }
  },
});

/**
 * MENU ITEM: Manual Trigger (For Debugging/Admins)
 * This creates a button in the subreddit menu visible only to moderators.
 * It allows forcing a "new day" event, essential for testing without waiting 24 hours.
 */
Devvit.addMenuItem({
  label: 'Force Market Open (Debug)',
  location: 'subreddit',
  forUserType: 'moderator',
  onPress: async (_event, context) => {
    try {
      const scenario = await generateDailyScenario(context);
      const dateKey = new Date().toISOString().split('T');
      await backupDailyState(context, dateKey, scenario);
      context.ui.showToast(`Market Force Opened: ${scenario.headline}`);
    } catch (e) {
      context.ui.showToast(`Error: ${e.message}`);
    }
  },
});

/**
 * POST TYPE: The Game Interface
 * This defines the Custom Post that users interact with.
 *
 * Rendering Strategy:
 * We use 'webview' to render a full React application. To ensure fast load times,
 * we fetch the initial state (the current market scenario) on the server side
 * and pass it into the webview's 'state' prop.
 */
Devvit.addCustomPostType({
  name: 'Get Rich Fast Simulator',
  height: 'tall', // "Tall" is required for the complex 3-panel layout
  render: (context) => {

    // Server-Side Data Fetching
    // The useState hook here runs on the server.
    const = useState(async () => {
      const dateKey = new Date().toISOString().split('T');

      // Fetch the shared market state
      const scenario = await getLatestScenario(context);

      // Fetch user-specific data (username)
      // Note: Redis access for user balance happens inside the app logic or via separate API
      const currentUser = context.userId? await context.reddit.getUserById(context.userId) :
null;

      return {
        date: dateKey,
        scenario: scenario,
        username: currentUser? currentUser.username : 'Anon',
      };
    });

    return (
      <webview
      id="game-view"
      url="page.html"
      onMessage={(msg) => {
        // This handler receives messages from the Webview (client).
        // It can be used for secure trade processing if we move logic server-side.
        console.log("Client message:", msg);
      }}
      state={initialData} // Hydrates the Webview with the fetched data
    />
   );
  },
});

export default Devvit;
```

---

4. Generative AI Integration: src/server/gemini.ts

The "intelligence" of "Get Rich Fast" is powered by the Google Gemini API. This module acts as the interface between the game's rigid logic and the creative chaos of the Large Language Model (LLM).

4.1 Model Selection and JSON Mode

The research indicates that the **Gemini 1.5 Flash** model is the optimal choice for this application.[14] The "Flash" designation implies a model optimized for low latency and cost-efficiency, which is crucial for a free-to-play game that might eventually scale to thousands of daily generations if the architecture changes to per-user scenarios.
Critically, the integration utilizes **JSON Mode** (Structured Outputs). By setting the responseMimeType to application/json in the API request, we coerce the model into returning a strictly formatted JSON object. This eliminates the need for fragile regex parsing of the model's output and ensures the game UI never breaks due to a malformed response.[16]

4.2 Prompt Engineering for "WallStreetBets" Tone

To satisfy the requirement for a "WallStreetBets" tone [1], the system instruction must be meticulously crafted. The prompt cannot simply ask for "financial news." It must explicitly instruct the model to adopt a specific persona—cynical, high-energy, and saturated with community-specific slang.
Key slang terms incorporated into the prompt engineering include:

- **Stonks:** Use of intentional misspellings to denote stocks.
- **Tendies:** Referring to gains/profits.
- **Diamond Hands:** Refusal to sell despite losses.

- **Paper Hands:** Weakness in selling early.
- **YOLO:** High-risk gambling on market movements.

The prompt also enforces the "Seinen/Noir" narrative by asking for "Noir-style descriptions" for the subtext, ensuring the text matches the visual aesthetic.

4.3 Source Code Specification: src/server/gemini.ts

TypeScript

```
import { Devvit } from '@devvit/public-api';

/**
 * Interface defining the strict structure of the Gemini response.
 * This ensures type safety throughout the application.
 */
export interface MarketScenario {
  headline: string;      // The "News" Panel: Punchy, uppercase, alarming.
  subtext: string;       // Context: Noir-style narration of the event.
  mood: 'BULL' | 'BEAR'; // Determines the UI color palette (Green/Red).
  advisor_vic: string;   // Vic's dialogue: High risk, "YOLO" attitude.
  advisor_sal: string;   // Sal's dialogue: Cynical, risk-averse, "Doomer".
  movement: number;      // The numerical impact on the market (e.g., +69, -420).
}

// Endpoint for Gemini 1.5 Flash
const GEMINI_API_URL =
'https://generativelanguage.googleapis.com/v1beta/models/gemini-1.5-flash:generateContent'
;

export async function generateDailyScenario(context: Devvit.Context):
Promise<MarketScenario> {

  // 1. Secure API Key Retrieval
  // The API key is stored in Reddit's Secret Store, not hardcoded.
  // Set via CLI: devvit settings set gemini_api_key <key>
  const apiKey = await context.settings.get('gemini_api_key');

  if (!apiKey) {
    throw new Error('Gemini API Key is missing! Configure it in app settings.');
  }

  // 2. Construct the Prompt
  // We use a "System Instruction" style prompt to define the persona and constraints.
  const prompt = `
    You are the "Dungeon Master" of a high-stakes, satirical stock market simulator called "Get
Rich Fast".
    Your aesthetic is "Noir Manga" and your tone is "r/WallStreetBets".

    Task: Generate a daily market event scenario.

    Constraints:
    1. Tone: Cynical, meme-heavy, high-energy. Use slang like "Stonks", "Tendies", "To the
Moon", "Paper Hands", "Wife's Boyfriend".
    2. Format: Return ONLY raw JSON. No markdown fencing.
    3. Characters:
       - Vic: A reckless suit. Always says "YOLO" or pushes for high risk. Represents Greed.
       - Sal: An old cynical trader. Remembers the crash of '29 and '08. Thinks everything is a
bubble. Represents Fear.

    JSON Schema Requirement:
    {
      "headline": "Short punchy headline (max 8 words)",
      "subtext": "A brief, noir-style description of the market chaos (max 20 words)",
      "mood": "BULL" (if good news) or "BEAR" (if bad news),
      "advisor_vic": "Vic's advice (max 15 words)",
      "advisor_sal": "Sal's advice (max 15 words)",
      "movement": Integer between -99 and +500 representing percentage change.
    }
  `;

  try {
    // 3. Execute HTTP Request
    const response = await fetch(`${GEMINI_API_URL}?key=${apiKey}`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        contents: [{ parts: [{ text: prompt }] }],
        generationConfig: {
           // This forces the model to output valid JSON, reducing parsing errors.
           responseMimeType: "application/json"
        }
      }),
    });

    if (!response.ok) {
      throw new Error(`Gemini API Error: ${response.status} ${response.statusText}`);
    }

    const data = await response.json();

    // 4. Extract and Validate Content
    const rawText = data.candidates?.?.content?.parts?.?.text;

    if (!rawText) {
      throw new Error('Empty response from Gemini');
    }

    const scenario: MarketScenario = JSON.parse(rawText);

    // Basic validation to ensure all fields exist
    if (!scenario.headline ||!scenario.mood) {
      throw new Error('Invalid JSON structure from Gemini');
    }

    return scenario;

  } catch (error) {
    console.error('Failed to generate scenario:', error);

    // 5. Fallback Mechanism
    // If the API fails, return a hardcoded "Market Halt" scenario to keep the game running.
    return {
      headline: "MARKET HALTED",
      subtext: "The exchange servers are smoking. Trading suspended.",
      mood: "BEAR",
      advisor_vic: "Let me in! Let me INNNN!",
      advisor_sal: "Told you the grid would fail.",
      movement: 0
    };
  }
}
```

---

5. Persistence Layer: src/data/backups.ts

While the filename backups.ts is requested, functionally this module serves as the Data Access Layer (DAL) for the application. It manages all interactions with Reddit's Redis instance.

5.1 Redis Data Modeling and Limitations

Reddit provides a Redis instance for data storage, but it comes with strict quotas: typically a **500MB storage limit** per installation.[6] This constraint necessitates a disciplined data modeling strategy.
The data model for "Get Rich Fast" utilizes a key-value structure optimized for two distinct access patterns:

1. **Read-Heavy Global State:** The daily scenario is read by every user every time they load the game. We cache this in a dedicated market:latest key for O(1) access complexity.
2. **Write-Heavy User State:** User balances are stored individually. To prevent "noisy neighbor" issues and allow for potential future leaderboards, we store user data with namespaced keys user:{id}:balance.

5.2 Data Archival and Expiry

To adhere to the storage limits, we implement an automatic expiration policy (TTL) on historical data. While the current day's scenario is kept indefinitely (until overwritten), historical scenarios stored in market:history:{date} are set to expire after 30 days. This "rolling window" approach ensures that the database does not grow unboundedly over time, which would eventually crash the app once the 500MB limit is reached.

5.3 Source Code Specification: src/data/backups.ts

TypeScript

```typescript
import { Devvit } from '@devvit/public-api';
import { MarketScenario } from '../server/gemini.js';

/**
 * Persists the daily scenario to Redis.
 * This function updates the 'current' state and archives the previous state.
 */
export async function backupDailyState(
  context: Devvit.Context,
  dateKey: string,
  scenario: MarketScenario
): Promise<void> {

  const serialized = JSON.stringify(scenario);
  // 1. Update the 'latest' pointer.
  // This is the key fetched by the Webview on load.
  await context.redis.set('market:latest', serialized);

  // 2. Archive the day's event for history features.
  // We verify strict TTL (Time To Live) to manage storage quotas.
  const historyKey = `market:history:${dateKey}`;
  await context.redis.set(historyKey, serialized);
  await context.redis.expire(historyKey, 60 * 60 * 24 * 30); // 30 day retention
}

/**
 * Retrieves the active scenario.
 * Includes a "Day 0" fallback if the database is empty.
 */
export async function getLatestScenario(context: Devvit.Context): Promise<MarketScenario> {
  const data = await context.redis.get('market:latest');

  if (!data) {
    // Default state for the very first run before the scheduler fires.
    return {
      headline: "WELCOME TO THE CASINO",
      subtext: "The market is waiting for the opening bell. Patience, degenerate.",
      mood: "BULL",
      advisor_vic: "Ready to bet the house?",
      advisor_sal: "Keep your money under the mattress.",
      movement: 0
    };
  }

  return JSON.parse(data);
}

/**
 * Updates a user's balance based on their action (YOLO vs FOLD).
 * This function encapsulates the core game logic for the simulation.
 */
export async function processTrade(
  context: Devvit.Context,
  userId: string,
  action: 'YOLO' | 'FOLD',
  marketMovement: number
): Promise<number> {
    const balanceKey = `user:${userId}:balance`;

    // Fetch current balance or initialize with starting capital ($1000)
    const currentBalanceStr = await context.redis.get(balanceKey);
    let balance = currentBalanceStr? parseInt(currentBalanceStr) : 1000;

    if (action === 'FOLD') {
      // FOLD logic: Player sits out.
      // Optional: Deduct a small "inflation fee" to discourage passivity?
      // For now, we return balance unchanged.
      return balance;
    }

    // YOLO Logic: Balance moves by the market percentage.
    // Leveraged logic: The movement is applied directly to the principal.
    const multiplier = 1 + (marketMovement / 100);
    balance = Math.floor(balance * multiplier);

    // Bankruptcy Protection:
    // In true WSB fashion, you can't go below $1 (need bus money).
    if (balance <= 0) balance = 1;

    // Persist new balance
    await context.redis.set(balanceKey, balance.toString());

    return balance;
}
```

---

6. Frontend Architecture: src/webview/App.tsx

The frontend is where the "Manga" aesthetic [7] is realized. We employ React, the standard framework for Devvit Webviews, alongside rigorous CSS Grid implementations to create the requested layout.

6.1 The "Manga" Layout Strategy

The layout is a direct translation of the visual requirements into CSS Grid. The interface is partitioned vertically into three rigidly defined zones:

1. **Panel 1: The News (Top):** This panel demands a "jagged" bottom border to mimic the torn paper or explosive action lines typical of manga. We achieve this using the clip-path CSS property with a polygon() function, creating a sawtooth pattern that no

standard border style can replicate.[19]

2. **Panel 2: The Advisor (Middle):** This section hosts the characters. The background must feature a "halftone" dot pattern, a staple of manga printing. We simulate this using CSS radial-gradient transparency layers.[7]
3. **Panel 3: The Action (Bottom):** A high-contrast control deck containing the two primary interaction points: "YOLO" and "FOLD".

6.2 State Management and Interaction

The React component receives its initial state (date, scenario, username) via props injected by the server-side render in main.ts. This ensures the UI renders immediately with content, vital for maintaining user engagement. Local state (balance, tradeResult) handles the immediate feedback loop of the game mechanics.

6.3 Source Code Specification: src/webview/App.tsx

TypeScript

```
import React, { useState, useEffect } from 'react';
import { MarketScenario } from '../server/gemini';
import { Characters } from './components/Characters';
import './App.css'; // Importing the CSS defined in section 6.4

interface AppProps {
  // Data passed from server logic
  date: string;
  scenario: MarketScenario;
  username: string;
}

export const App: React.FC<AppProps> = ({ date, scenario, username }) => {
  // Local state for the interaction simulation
  const = useState<string | null>(null);

  // The 'Mood' determines the primary accent color across the UI.
  // We stick to a strict 2-color signal palette:
  // BULL = Terminal Green (#00FF41)
  // BEAR = Alert Red (#FF0033)
  const accentColor = scenario.mood === 'BULL'? '#00FF41' : '#FF0033';

  /**
   * Handling the core game action.
```

```
 * In a full implementation, this would send a message back to the Devvit server
 * to persist the new balance via context.redis.
 */
const handleAction = (action: 'YOLO' | 'FOLD') => {
  // Simulate the result for immediate feedback
  if (action === 'FOLD') {
     setTradeResult("You kept your cash. Boring, but safe.");
  } else {
     const gain = scenario.movement > 0;
     setTradeResult(
        gain
       ? `GAINS! You're eating tendies tonight! (+${scenario.movement}%)`
        : `GUH. You lost it all. (-${Math.abs(scenario.movement)}%)`
     );
  }
};

return (
  <div
    className="manga-container"
    style={{ '--accent': accentColor } as React.CSSProperties}
  >

    {/* PANEL 1: THE NEWS */}
    {/* Uses the clip-path border effect */}
    <div className="panel-news manga-border-bottom">
      <div className="header-row">
         <span className="news-date">VOL. {date}</span>
         <span className="news-ticker">GRF-SIM</span>
      </div>

      <h1 className="news-headline">{scenario.headline}</h1>
      <p className="news-subtext">{scenario.subtext}</p>

      <div className="market-movement" style={{ color: accentColor }}>
         MARKET: {scenario.movement > 0? '+' : ''}{scenario.movement}%
      </div>
    </div>

    {/* PANEL 2: THE ADVISOR */}
    {/* Uses the halftone background effect */}
    <div className="panel-advisor manga-bg">
      <Characters
```

```
          mood={scenario.mood}
          vicQuote={scenario.advisor_vic}
          salQuote={scenario.advisor_sal}
          accentColor={accentColor}
        />
      </div>

      {/* PANEL 3: THE ACTION */}
      <div className="panel-action">
        {!tradeResult? (
          <div className="button-grid">
            <button
              className="btn-yolo"
              onClick={() => handleAction('YOLO')}
              style={{ borderColor: accentColor, color: accentColor }}
            >
              YOLO
            </button>
            <button
              className="btn-fold"
              onClick={() => handleAction('FOLD')}
            >
              FOLD
            </button>
          </div>
        ) : (
          <div className="result-display" style={{ borderColor: accentColor }}>
            <h2 style={{ color: accentColor }}>{tradeResult}</h2>
            <p>Check back tomorrow for the next opening bell.</p>
          </div>
        )}
      </div>
    </div>
  );
};
```

6.4 CSS Implementation: src/webview/App.css

The following CSS is critical for the visual identity and must be included in the project. It handles the clip-path geometry and the halftone generation.

CSS

```
:root {
  --black: #111;
  --white: #fff;
  --accent: #00FF41; /* Default, overridden by React inline style */
}

/* Base Layout */
.manga-container {
  display: grid;
  grid-template-rows: auto 1fr 140px; /* News fits content, Advisors fill space, Action fixed
height */
  height: 100vh;
  background: var(--white);
  font-family: 'Courier New', Courier, monospace; /* Monospace for that "Terminal/Document"
feel */
  color: var(--black);
  overflow: hidden;
}

/* Panel 1: News */
.panel-news {
  padding: 20px;
  background: var(--white);
  border-bottom: 5px solid var(--black);
  position: relative;
  z-index: 10;

  /* The jagged edge effect at the bottom */
  /* This polygon defines a sawtooth pattern along the bottom edge (95%-100% Y) */
  clip-path: polygon(
    0% 0%, 100% 0%, 100% 95%,
    95% 100%, 90% 95%, 85% 100%, 80% 95%, 75% 100%, 70% 95%, 65% 100%,
    60% 95%, 55% 100%, 50% 95%, 45% 100%, 40% 95%, 35% 100%, 30% 95%,
    25% 100%, 20% 95%, 15% 100%, 10% 95%, 5% 100%, 0% 95%
  );
}

.news-headline {
  font-size: 2.5rem;
  font-weight: 900;
  text-transform: uppercase;
  margin: 10px 0;
  line-height: 0.9;
  letter-spacing: -2px;
}

/* Panel 2: Advisor */
.panel-advisor {
  position: relative;
  display: flex;
  align-items: center;
  justify-content: center;
}

.manga-bg {
  /* Creating the Halftone (Dot) Pattern */
  /* Two radial gradients offset by 5px create the interlocking dot matrix */
  background-image:
    radial-gradient(var(--black) 20%, transparent 20%),
    radial-gradient(var(--black) 20%, transparent 20%);
  background-color: var(--white);
  background-position: 0 0, 5px 5px;
  background-size: 10px 10px;
  opacity: 0.15; /* Subtle texture, not overwhelming */
}

/* Panel 3: Action */
.panel-action {
  background: var(--black);
  padding: 10px;
  display: flex;
  align-items: center;
  justify-content: center;
}

.button-grid {
  display: grid;
  grid-template-columns: 1fr 1fr;
  gap: 20px;
  width: 100%;
  max-width: 600px;
}

button {
  height: 80px;
  font-family: inherit;
  font-weight: 900;
  font-size: 1.5rem;
  text-transform: uppercase;
  border: 4px solid;
  cursor: pointer;
  transition: transform 0.1s;
}

button:active {
  transform: scale(0.95);
}

.btn-yolo {
  background: var(--black);
  /* Accent color provided by inline style from React */
}

.btn-fold {
  background: var(--white);
  color: var(--black);
  border-color: var(--white);
}

.result-display {
  color: var(--white);
  text-align: center;
  border: 2px dashed;
  padding: 20px;
  width: 100%;
}
```

---

7. Visual Components: src/webview/components/Characters.tsx

The visual representation of the characters "Vic" and "Sal" relies on SVG manipulation rather than raster images. This approach is superior for the Devvit environment as it keeps the bundle size small and allows for dynamic recoloring based on the mood variable without needing to load separate assets.[7]

7.1 SVG Path Selection and Logic

We utilize standard icon geometries that are publicly available to construct the silhouettes.

- **Vic (The Bull):** Represented by a "Man in Suit" silhouette. This captures the "Wall Street" persona—clean lines, tie, professional but potentially reckless.
- **Sal (The Bear):** Represented by a "Detective/Spy" silhouette (Hat and Trenchcoat). This visual shorthand communicates mystery, cynicism, and age ("Old Man").[22]

7.2 Source Code Specification: src/webview/components/Characters.tsx

TypeScript

```
import React from 'react';

interface CharactersProps {
  mood: 'BULL' | 'BEAR';
  vicQuote: string;
  salQuote: string;
  accentColor: string;
}

export const Characters: React.FC<CharactersProps> = ({ mood, vicQuote, salQuote,
accentColor }) => {

  // Vic Path: Standard "User Tie" geometry
  const vicPath = "M224 256c70.7 0 128-57.3 128-128S294.7 0 224 0 96 57.3 96 128s57.3 128 128
128zm89.6 32h-16.7c-22.2 10.2-46.9 16-72.9 16s-50.6-5.8-72.9-16h-16.7C60.2 288 0 348.2 0
422.4V464c0 26.5 21.5 48 48 48h352c26.5 0 48-21.5
48-48v-41.6c0-74.2-60.2-134.4-134.4-134.4z";

  // Sal Path: "User Secret" geometry (Hat/Coat)
  const salPath = "M224 256c70.7 0 128-57.3 128-128S294.7 0 224 0 96 57.3 96 128s57.3 128 128
128zm89.6 32h-16.7c-22.2 10.2-46.9 16-72.9 16s-50.6-5.8-72.9-16h-16.7C60.2 288 0 348.2 0
422.4V464c0 26.5 21.5 48 48 48h352c26.5 0 48-21.5
48-48v-41.6c0-74.2-60.2-134.4-134.4-134.4z";

  // Note regarding SVG Paths:
  // In a production environment, exact path data from FontAwesome or similar
  // open libraries would be pasted here. The strings above are illustrative
  // placeholders for the complex coordinate data required for high-fidelity icons.
```

```
return (
 <div className="characters-grid" style={{
    display: 'grid',
    gridTemplateColumns: '1fr 1fr',
    width: '100%',
    height: '100%',
    alignItems: 'end',
    paddingBottom: '20px'
 }}>

  {/* VIC (Left) - The Aggressor */}
  <div className="char-container vic" style={{ textAlign: 'center', position: 'relative' }}>
   {/* Speech Bubble: Vic's advice */}
   <div className="speech-bubble" style={{
      background: '#fff',
      border: `3px solid ${accentColor}`,
      padding: '10px',
      marginBottom: '10px',
      borderRadius: '10px 10px 10px 0',
      fontWeight: 'bold',
      boxShadow: '4px 4px 0px #000'
   }}>
      "{vicQuote}"
   </div>

   {/* Vic Avatar */}
   <svg viewBox="0 0 448 512" height="180" width="180">
     <path d={vicPath} fill={mood === 'BULL'? accentColor : '#333'} />
   </svg>
   <div className="char-name" style={{ fontWeight: 900, marginTop: '5px' }}>VIC</div>
  </div>

  {/* SAL (Right) - The Pessimist */}
  <div className="char-container sal" style={{ textAlign: 'center', position: 'relative' }}>
   {/* Speech Bubble: Sal's advice */}
   <div className="speech-bubble" style={{
      background: '#fff',
      border: '3px solid #000',
      padding: '10px',
      marginBottom: '10px',
      borderRadius: '10px 10px 0 10px',
      fontStyle: 'italic',
          boxShadow: '-4px 4px 0px #000'
        }}>
          "{salQuote}"
        </div>

        {/* Sal Avatar */}
        <svg viewBox="0 0 448 512" height="180" width="180">
          <path d={salPath} fill={mood === 'BEAR'? accentColor : '#333'} />
        </svg>
        <div className="char-name" style={{ fontWeight: 900, marginTop: '5px' }}>SAL</div>
      </div>

    </div>
  );
};
```

---

8. Deployment Strategy and Testing

Deploying a serverless application with scheduled components requires a specific sequence of operations to ensure all dependencies (specifically the external API keys) are present before the code begins execution.

8.1 Pre-Deployment Checklist

1. **API Key Configuration:** The Gemini API key must be securely stored in the app's settings. This is done via the CLI command:
   devvit settings set gemini_api_key "YOUR_GOOGLE_API_KEY"
2. **Asset Handling:** While this implementation relies on SVGs, if any static raster assets were to be added, they would need to be placed in the assets/ directory and declared in devvit.json.

8.2 Testing the "Daily" Loop

Testing a 24-hour cycle is impractical during development. This is why the main.ts file includes the "Force Market Open" menu item. The recommended testing workflow is:

1. Run devvit playtest <subreddit>.
2. Navigate to the subreddit as a moderator.
3. Trigger "Force Market Open" from the subreddit menu.
4. Verify in the logs that:
   - The Gemini API was called.
   - The JSON was parsed correctly.
   - The Redis keys (market:latest) were updated.

5. Load the custom post and verify the Webview hydrates with the new content.

8.3 Error Handling and Resilience

The system is designed with a "fail-open" philosophy. If the Gemini API fails (quota limits, outage), the generateDailyScenario function catches the error and returns a hardcoded "Market Halt" scenario. This ensures that the scheduled job never crashes the application entirely, and users always see *some* valid state when they load the game.

9. Conclusion

"Get Rich Fast" illustrates the potential of the Reddit Devvit platform to host complex, stateful narratives without traditional infrastructure. By strictly adhering to a serverless pattern, leveraging Redis for state continuity, and utilizing Generative AI for endless content variety, this architecture delivers a high-engagement experience with minimal operational overhead. The rigorous application of the "Manga" aesthetic through CSS and SVG manipulation further ensures the product stands out in the feed, satisfying the unique cultural expectations of its target audience.

Works cited

1. Ultimate Guide to WallStreetBets Terms and Lingo | SWFI, accessed February 3, 2026, https://www.swfinstitute.org/news/84215/ultimate-guide-to-wallstreetbets-terms-and-lingo
2. The Ultimate Guide to Reddit's Wallstreetbets Slang - Infinity Investing, accessed February 3, 2026, https://infinityinvesting.com/wallstreetbets-slang-meaning/
3. Building Reddit Game with Devvit and TypeScript (starter included) - DEV Community, accessed February 3, 2026, https://dev.to/room_js/building-reddit-game-with-devvit-and-typescript-starter-included-3kcp
4. @devvit/server | Reddit for Developers, accessed February 3, 2026, https://developers.reddit.com/docs/0.11/devvit_web/server
5. Scheduler - Reddit for Developers, accessed February 3, 2026, https://developers.reddit.com/docs/capabilities/server/scheduler
6. Redis - Reddit for Developers, accessed February 3, 2026, https://developers.reddit.com/docs/capabilities/server/redis
7. Context_ Manga UI Architecture.pdf
8. A Responsive Comic Panel Layout with CSS Grid - Nevin Katz, accessed February 3, 2026, https://nevkatz.github.io/2020/01/06/css-grid-intro.html
9. Z-Shaped Pattern For Reading Web Content | by Nick Babich | UX Planet, accessed February 3, 2026, https://uxplanet.org/z-shaped-pattern-for-reading-web-content-ce1135f92f1c
10. HTTP Fetch - Reddit for Developers, accessed February 3, 2026, https://developers.reddit.com/docs/capabilities/server/http-fetch
11. Automating Daily Interactive Posts with Dynamic Images on Devvit - Reddit,

accessed February 3, 2026, https://www.reddit.com/r/Devvit/comments/1ojdk2i/automating_daily_interactive_posts_with_dynamic/

12. Working with useState - Reddit for Developers, accessed February 3, 2026, https://developers.reddit.com/docs/0.11/working_with_usestate
13. Web views - Reddit for Developers, accessed February 3, 2026, https://developers.reddit.com/docs/0.11/webviews
14. Gemini 3.0 Flash: Google's Greatest Model Ever? Most Powerful, Cheapest, & Fastest Model! (Tested), accessed February 3, 2026, https://www.youtube.com/watch?v=izXjYxKTI_k
15. Learn about supported models | Firebase AI Logic - Google, accessed February 3, 2026, https://firebase.google.com/docs/ai-logic/models
16. How to get JSON output from gemini-1.5-pro-001 using curl - Stack Overflow, accessed February 3, 2026, https://stackoverflow.com/questions/78779183/how-to-get-json-output-from-gemini-1-5-pro-001-using-curl
17. Generate structured output (like JSON and enums) using the Gemini API | Firebase AI Logic, accessed February 3, 2026, https://firebase.google.com/docs/ai-logic/generate-structured-output
18. What are you using for storage? : r/Devvit - Reddit, accessed February 3, 2026, https://www.reddit.com/r/Devvit/comments/1qktf3v/what_are_you_using_for_storage/
19. Paths, shapes, clipping, and masking - CSS - web.dev, accessed February 3, 2026, https://web.dev/learn/css/paths-shapes-clipping-masking
20. Using CSS Masks to Create Jagged Edges, accessed February 3, 2026, https://css-tricks.com/using-css-masks-to-create-jagged-edges/
21. Person standing dress · Bootstrap Icons, accessed February 3, 2026, https://icons.getbootstrap.com/icons/person-standing-dress/
22. Men in Suits Silhouette Svg - Etsy, accessed February 3, 2026, https://www.etsy.com/market/men_in_suits_silhouette_svg?ref=lp_queries_internal_bottom-5
23. Old man with hat silhouette Images - Free Download on Freepik, accessed February 3, 2026, https://www.freepik.com/free-photos-vectors/old-man-with-hat-silhouette