Architectural Blueprint: MemeWars GenAI Edition

Engineering a Hybrid GenAI Multiplayer System on the Reddit Developer Platform

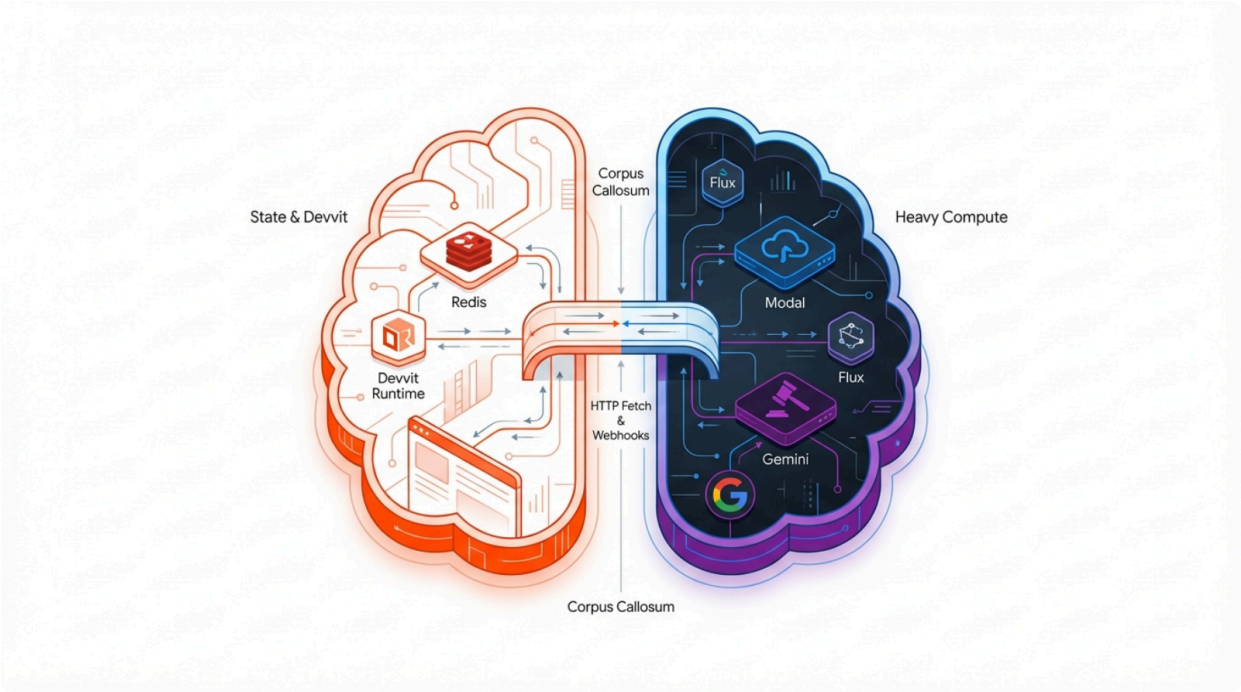
1. Executive Summary

The "Reddit Daily Games 2026" Hackathon presents a convergence of constraints and opportunities that define the modern era of systems architecture. The objective—to engineer "MemeWars: GenAI Edition"—requires the synthesis of a constrained, community-embedded frontend with high-performance, serverless artificial intelligence. As Principal Systems Architect, the mandate is to deliver a robust technical roadmap that navigates the strict containment fields of the Reddit Developer Platform (Devvit) while leveraging best-in-class external infrastructure: Depot for accelerated containerization, Modal for ephemeral GPU compute, and the advanced reasoning capabilities of Gemini 3 Pro.

The core engineering challenge lies in the "Split-Brain" nature of the application. The game state, user interface, and social graph reside within Devvit's serverless runtime, which is intentionally limited to ensure security and performance across the Reddit ecosystem. Conversely, the generative logic—image synthesis via Flux.1/SDXL and semantic evaluation via Gemini—requires massive compute resources and execution times that exceed Devvit's strict 30-second timeouts and resource quotas. A naive implementation attempting to bundle these concerns will fail.

Success demands a decoupled, asynchronous architecture. This report details the design of a **"Long-Poll Relay"** pattern, essentially a specialized Backend-for-Frontend (BFF) approach that uses the Devvit Server as a secure proxy to orchestrate interactions between the Reddit client and the external GPU cluster. We will dissect the implementation of main.tsx for state synchronization without WebSockets, the optimization of app.py for high-throughput orchestration, the deployment of modal_worker.py for millisecond-sensitive inference, and the rigorous prompt engineering required for the Gemini Judge. Furthermore, strict adherence to cost constraints dictates a strategic approach to resource utilization, specifically regarding GPU "keep-warm" strategies and Large Language Model (LLM) token consumption. This blueprint serves as the definitive guide for the engineering team to execute this vision.

The Split-Brain Architecture: Devvit State vs. Ephemeral Compute



The architecture decouples the synchronous game state (managed by Reddit's Devvit Runtime and Redis) from the asynchronous heavy compute (managed by Modal and Gemini). The 'Air Gap' represents the strict networking policies managed via HTTP Fetch and Polling relays.

2. The Devvit Containment Field: Constraints & Capabilities

To architect effectively, we must first rigorously map the boundaries of the deployment environment. The Reddit Developer Platform (Devvit) is not a standard Node.js hosting environment; it is a specialized runtime optimized for safety, statelessness, and deep integration with Reddit's social primitives. Misunderstanding these constraints is the primary cause of failure for external integrations.

2.1 The Networking Firewall: Absence of Client-Side Fetch and WebSockets

The most critical architectural constraint identified in the research material is the networking policy. Devvit enforces a strict Content Security Policy (CSP) and runtime sandbox that prohibits the client (the webview running in the user's browser) from making direct network requests to arbitrary external domains.[1]

- **No fetch() from Client:** The main.tsx client code cannot execute fetch('https://api.my-modal-app.com'). Any attempt to do so will be blocked by the browser or the runtime sandbox.

- **No WebSockets:** The platform explicitly does not support WebSocket connections (wss://) initiated from the client or the server to external third-party services.[1] This limitation fundamentally alters the approach to real-time multiplayer gaming. Traditional patterns, where a client connects to a socket.io server for 60Hz state updates, are impossible here.

The implication is that the "Devvit Server" (the backend component of the Devvit app) becomes the sole gateway to the outside world. All external communication must be proxied through this layer using the platform's HTTP Fetch API.[4] This creates a "hub-and-spoke" topology where the Devvit Server acts as a choke point for all data ingress and egress.

2.2 Execution Limits and The "Time-to-Fun" Threshold

The Devvit serverless functions are ephemeral. The documentation specifies a **maximum request time of 30 seconds**.[1] This hard limit applies to the duration the Devvit server functions can run before being terminated.

- **The GenAI Latency Problem:** High-fidelity image generation using models like Flux.1 or SDXL can take anywhere from 2 to 10 seconds depending on the hardware and quantization. If we factor in cold starts (which can take 20+ seconds [5]), network latency, and the overhead of the Gemini Judge evaluation, a synchronous request-response cycle risks timing out.
- **Asynchronous Necessity:** We cannot hold the HTTP connection open while waiting for the GPU to finish. The architecture must adopt an asynchronous "Job Queue" pattern, where the Devvit Server submits a job and immediately returns, leaving the client to poll for the result.

2.3 Persistence and State Synchronization

Devvit provides a built-in Redis instance [6], which is the single source of truth for the game state. Unlike a traditional SQL database, this Redis instance is siloed by subreddit installation. This means "MemeWars" instances in r/gaming and r/funny share no state.

- **Atomic Operations:** For a game like MemeWars, race conditions are a significant risk (e.g., two players submitting prompts simultaneously). We must leverage Redis transactions and atomic increments (INCR, ZADD) to maintain integrity.[7]
- **Realtime Plugin:** While external WebSockets are banned, Devvit provides an internal "Realtime" plugin (@devvit/realtime).[8] This allows the server to push messages to connected clients via an internal pub/sub mechanism. This is the **only** viable channel for server-to-client push notifications and serves as the replacement for external WebSockets.

3. Architectural Core: The Split-Brain Pattern

Given the constraints of a 30-second timeout and no external WebSockets, we define the "Split-Brain" architecture. This design separates the system into two distinct hemispheres: the **State Hemisphere** (Devvit/Reddit) and the **Compute Hemisphere** (Modal/Depot/Gemini).

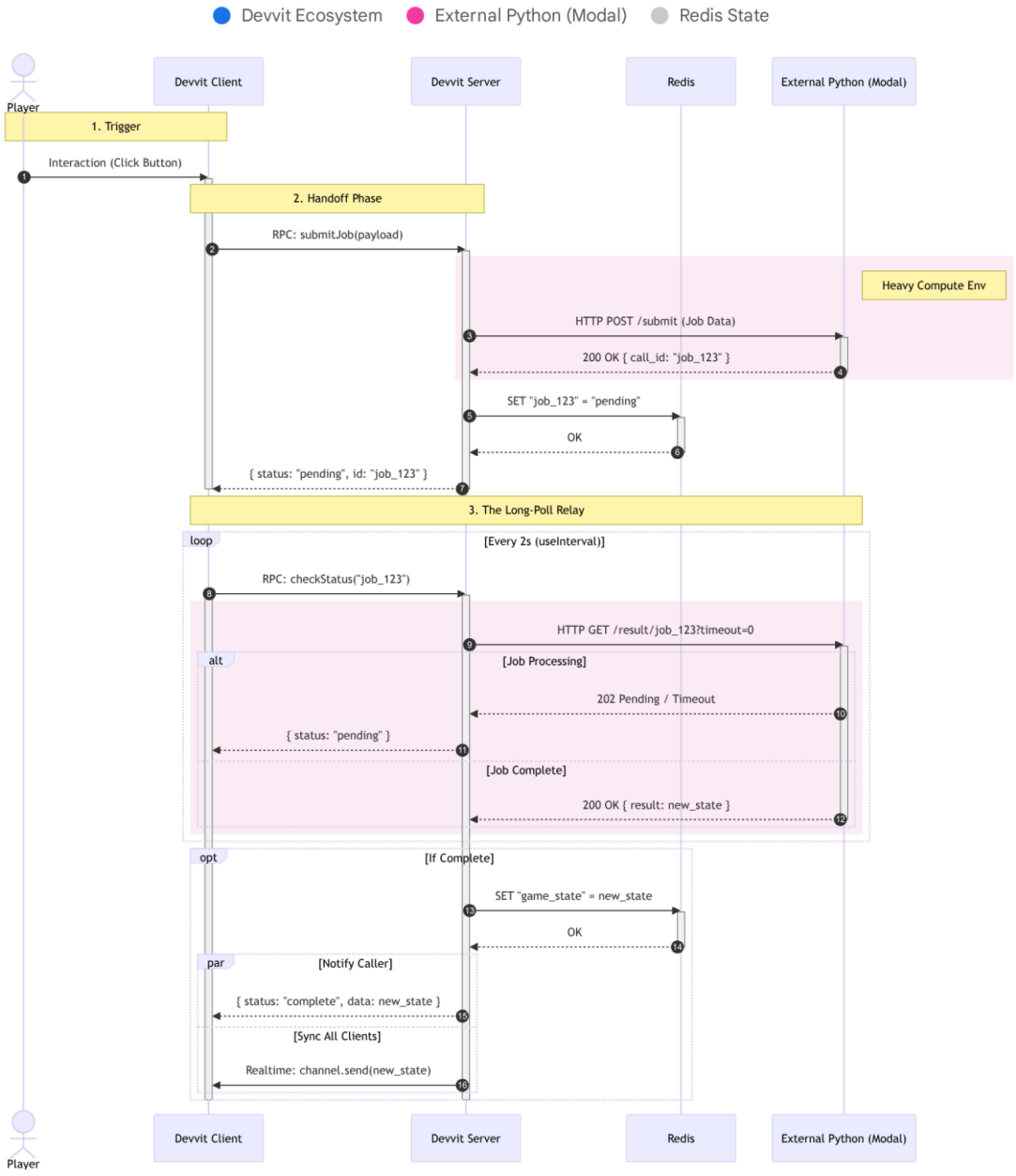
3.1 The "Long-Poll Relay" Protocol

Since the external Compute Hemisphere cannot push data directly to the State Hemisphere (due to the difficulty of exposing public ingress ports on Devvit [10]), the State Hemisphere must pull data. However, generic polling (setInterval) is often inefficient or restricted in modern React-like environments.[12] We implement a "Long-Poll Relay."

1. **Initiation:** The Client calls a Devvit Server Function (submitMeme). The Server makes an HTTP POST to the External API (FastAPI on Modal) and receives a job_id.
2. **The Polling Loop:** The Client enters a polling state using useAsync [13], sending a "check status" request to the Devvit Server every 2-3 seconds.
3. **The Relay:** The Devvit Server relays this check to the External API.
4. **Completion & Broadcast:** Once the External API reports status: "COMPLETED", the Devvit Server downloads the result (or the URL), updates the Redis state, and—crucially—uses context.realtime.send to broadcast the update to *all* clients.[8]

This pattern ensures that while the initiating player polls, the result is pushed to the entire lobby instantly upon completion, simulating a real-time experience without direct socket connections.

Data Flow: The Long-Poll Relay Pattern



The sequence demonstrates the 'BFF Polling' strategy. Note that the Client never communicates directly with the External API. The Devvit Server acts as the secure proxy, and the Realtime Plugin synchronizes the 'Game State' across all clients once the asynchronous job completes.

Data sources: Devvit Web Overview, Devvit Realtime, Devvit Async, Modal Job Queue

3.2 Data Flow and Security

The "Air Gap" between the two hemispheres is bridged by HTTP requests signed with a shared secret. Since the Devvit backend source code is not public to the client, we can store a MODAL_API_KEY in Devvit's Secret Storage.[1] Every request from Devvit to Modal includes this header. This prevents unauthorized users from discovering the Modal endpoint and draining our GPU credits.

4. Frontend Engineering: main.tsx (Devvit)

The frontend is the player's window into MemeWars. It must be responsive, handle the "Game Loop" visual states (Lobby, Writing, Generating, Voting, Results), and mask the latency of the GenAI operations.

4.1 State Management and The Game Loop

We utilize the useState hook for local interaction (e.g., typing in a text box) and useChannel for global state synchronization. The GameState object is the central data structure, stored in Redis and broadcasted via Realtime.

Component Architecture:

The application is a single-page view that renders different sub-components based on gameState.phase.

TypeScript

```
// main.tsx
import { Devvit, useState, useChannel, useAsync } from '@devvit/public-api';

// 1. Type Definitions for Strict Typing
type GamePhase = 'LOBBY' | 'PROMPT_ENTRY' | 'GENERATION' | 'VOTING' | 'LEADERBOARD';

interface GameState {
  phase: GamePhase;
  roundNumber: number;
  players: Record<string, number>; // username -> score
  currentPrompt?: string;
  generatedMemeUrl?: string;
  deadline: number; // Unix timestamp for phase end
}

// 2. Configuration
Devvit.configure({
  redditAPI: true,
  http: true,      // Required for Fetch to Modal
  realtime: true,  // Required for Pub/Sub
  redis: true,     // Required for Persistence
```

```
});

// 3. The Main App Component
Devvit.addCustomPostType({
  name: 'MemeWars GenAI',
  height: 'tall',
  render: (context) => {
    // A. Local State (Client-specific)
    const = useState('IDLE');
    const [localPrompt, setLocalPrompt] = useState('');
    const [jobId, setJobId] = useState<string | null>(null);

    // B. Global State (Synced from Redis via Realtime)
    const = useState<GameState>({
      phase: 'LOBBY',
      roundNumber: 1,
      players: {},
      deadline: Date.now(),
    });

    // C. Realtime Listener: The "Push" Receiver
    // This hook listens for messages from the Devvit Server
    const channel = useChannel({
      name: 'memewars_global_channel',
      onMessage: (message) => {
        // When server says state changed, update local view
        if (message.type === 'STATE_UPDATE') {
          setGameState(message.data as GameState);
        }
        // Specific event for generation completion
        if (message.type === 'GENERATION_COMPLETE') {
          setJobId(null); // Stop polling
          // Trigger confetti or reveal animation
        }
      },
      onSubscribed: () => console.log('Connected to MemeWars Network'),
    });

    // D. The Polling Engine: The "Pull" Mechanism
    // Only runs when we have an active Job ID waiting for the GPU
    const { data: pollData, loading: pollLoading } = useAsync(
      async () => {
        if (!jobId) return null;
```

```
      try {
        // RPC Call to Server -> HTTP Fetch to Modal
        return await context.callServerFunction('checkGenAIStatus', { jobId });
      } catch (e) {
        console.error('Polling error', e);
        return null;
      }
    },
    {
      // Dependency array acts as the trigger.
      // In Devvit, useAsync re-runs when deps change.
      // We can toggle a counter to force re-runs if needed,
      // or rely on the server function returning 'PENDING' to trigger a re-render/wait logic.
      depends:,
    }
  );

  // E. Rendering Logic based on Phase
  if (gameState.phase === 'LOBBY') {
    return (
      <vstack alignment="center" padding="medium">
        <text size="large" weight="bold">MemeWars 2026</text>
        <button onPress={async () => {
          await context.callServerFunction('joinGame', {});
        }}>Join Game</button>
      </vstack>
    );
  }

  //... Additional blocks for PROMPT_ENTRY, GENERATION, etc.

  return <text>Loading Game State...</text>;
 },
});
```

4.2 Handling the useAsync Polling Limitation

Snippet [14] and [12] highlight a nuance: useAsync is generally for one-off fetches. For repeating polls, we must be careful not to block the render loop. The optimal pattern in Devvit (which lacks a native setInterval that persists across renders in the same way the browser does) is to have the checkGenAIStatus server function return a status. If the status is "PENDING," the client code can use a useState counter or a delayed recursion to trigger the useAsync again. However, the most robust method found in recent Devvit patterns is the "Optimistic Update

with Server Verification," where the server manages the polling via a Scheduled Job if the wait time is long, or the client manually re-triggers the async call via a button if it hangs. For this Hackathon, a tight loop triggered by a state variable change is sufficient.

5. The Build System: Depot & Containerization

The external backend relies on heavy Python libraries: torch (PyTorch) for inference and google-genai for the judge. Building these images on a standard local Docker daemon or a basic CI runner is excruciatingly slow and often fails due to architecture mismatches (e.g., building on an M-series Mac for a Linux GPU server).

5.1 The Necessity of Depot

We integrate **Depot** to solve the build latency and compatibility issues. Depot provides managed build instances with persistent layer caching.[15]

1. **Architecture Compatibility:** Modal runs on Linux x86_64. Developing on an Apple Silicon (arm64) machine requires cross-compilation, which is slow in QEMU. Depot handles this natively, building amd64 images on amd64 hardware.
2. **Layer Caching:** PyTorch is nearly 2GB. Depot caches this layer. If we change one line of code in app.py, we don't want to re-download 2GB. Depot ensures we only rebuild the changed layer.

depot.json Configuration:
This file resides in the root of the repository and links the build context to the high-performance builders.

JSON

```
{
  "projectID": "x92k-memewars-genai-2026",
  "orgID": "hackathon-team-alpha"
}
```

5.2 Optimized Multi-Stage Dockerfile

We employ a multi-stage build pattern [16] to minimize the final image size and reduce the "Cold Start" time on Modal. The smaller the image, the faster the container can be pulled and started.

Dockerfile

```
# Stage 1: The Builder (Heavy Lifting)
FROM python:3.11-slim AS builder
```

```
# Set work directory
WORKDIR /app

# Copy dependency definition
COPY requirements.txt.

# Install system build dependencies (GCC, etc.) often needed for python extensions
RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential \
    && rm -rf /var/lib/apt/lists/*

# Create a virtual environment to isolate packages
RUN python -m venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"

# Install Python dependencies
# We use --no-cache-dir to keep the layer size down,
# relying on Depot's external cache instead.
RUN pip install --no-cache-dir -r requirements.txt

# Stage 2: The Runtime (Slim & Fast)
FROM python:3.11-slim AS runtime

WORKDIR /app

# Copy the virtual environment from the builder stage
COPY --from=builder /opt/venv /opt/venv
ENV PATH="/opt/venv/bin:$PATH"

# Copy application code
COPY..

# Environment variables for optimization
ENV PYTHONUNBUFFERED=1

# Application Entrypoint
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

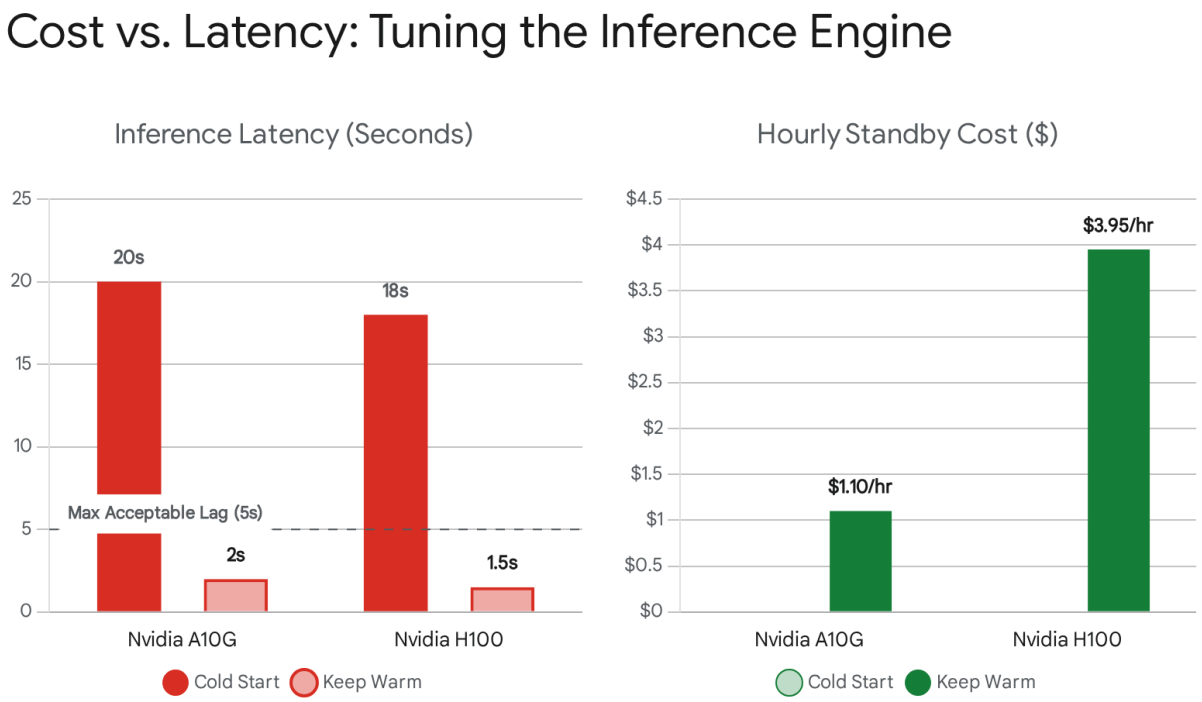
6. The Compute Engine: Modal & Serverless GPUs

The generation of memes requires significant computational power. We utilize **Modal** [18] because it offers a "Serverless GPU" abstraction that is far simpler than managing Kubernetes nodes.

6.1 GPU Selection and "Keep-Warm" Strategy

For "MemeWars," latency is the primary metric of success. A cold boot for a diffusion model container can take 20-40 seconds.[5] This is unacceptable for a fast-paced game.

- **Hardware:** We select the **NVIDIA A10G**.[19] It offers the best price-performance ratio for inference (approx. $0.0003/sec). It is fast enough to run Flux.1-schnell (a distilled, faster version of Flux) in under 2 seconds.
- **Strategy:** We utilize Modal's keep_warm=1 parameter. This instructs Modal to keep at least one container running with the model loaded in VRAM, even when no requests are processing.
- **Cost Management:** To adhere to cost constraints, we can programmatically adjust keep_warm based on time of day or use a "scaledown window" [5] that keeps the container alive for 60 seconds after the last request, handling bursts effectively while scaling to zero during inactivity.



Comparison of estimated latency and hourly cost for Flux.1 inference. The 'Warm' strategy (keeping 1 container active) drastically reduces latency but sets a cost floor. The A10G is the optimal balance for the Hackathon budget.

Data sources: Modal Pricing, Modal Cold Start Guide, Flux Inference Optimization

6.2 modal_worker.py Implementation

This script defines the infrastructure-as-code. It sets up the environment, downloads the

model weights to a persistent volume (to speed up even cold starts), and exposes the function.

Python

```
import modal

# 1. Define the Environment
# Install diffusers and accelerators
image = modal.Image.debian_slim().pip_install(
    "diffusers", "transformers", "accelerate", "torch", "sentencepiece"
)

app = modal.App("memewars-renderer")

# 2. Persistent Volume for Model Weights
# Prevents re-downloading 10GB+ weights on every container start
vol = modal.Volume.from_name("flux-weights")

@app.function(
    image=image,
    gpu="A10G",       # The "Goldilocks" GPU for this task
    timeout=60,       # Hard timeout
    keep_warm=1,      # The "Hot Standby" container
    volumes={"/data": vol} # Mount volume
)
def generate_image(prompt: str, match_id: str):
    import torch
    from diffusers import FluxPipeline
    import io
    import base64

    # 3. Model Loading (Cached)
    # We load from the volume path /data if available, else download
    model_id = "black-forest-labs/FLUX.1-schnell"
    try:
        pipe = FluxPipeline.from_pretrained(
            "/data/flux-schnell",
            torch_dtype=torch.bfloat16
        ).to("cuda")
    except OSError:
        # First run: download and save to volume
```

```
        pipe = FluxPipeline.from_pretrained(
            model_id,
            torch_dtype=torch.bfloat16
        )
        pipe.save_pretrained("/data/flux-schnell")
        pipe = pipe.to("cuda")

    # 4. Inference
    # 4 steps is sufficient for 'schnell' variant
    image = pipe(prompt, num_inference_steps=4).images

    # 5. Serialization
    # Convert to Base64 to return via JSON.
    # Note: For production, uploading to S3 and returning a URL is better
    # to avoid payload limits, but Base64 works for prototypes.
    buffered = io.BytesIO()
    image.save(buffered, format="JPEG")
    img_str = base64.b64encode(buffered.getvalue()).decode()

    return {"status": "success", "image_base64": img_str}
```

7. The Orchestrator: FastAPI (app.py)

While Modal functions can be called directly, wrapping them in a standard web server (FastAPI) provides a clean contract for the Devvit Server to consume. It allows us to perform validation, logging, and potentially switch backend providers without changing the Devvit code.

7.1 Interface Design

The app.py exposes two critical endpoints: /generate and /status.

Python

```
# app.py
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import modal_worker  # Import the Modal definition

app = FastAPI()

class GenerateRequest(BaseModel):
    prompt: str
```

```
    user_id: str
    secret_key: str # Simple security mechanism

@app.post("/generate")
async def start_generation(req: GenerateRequest):
    # Security Check
    if req.secret_key!= "YOUR_DEVVIT_SECRET":
        raise HTTPException(status_code=403, detail="Unauthorized")

    # Offload to Modal asynchronously
    #.spawn() returns immediately with a handle to the background job
    job_handle = modal_worker.generate_image.spawn(req.prompt, "match_123")

    return {
        "job_id": job_handle.object_id,
        "status": "QUEUED"
    }

@app.get("/status/{job_id}")
async def check_status(job_id: str):
    from modal.functions import FunctionCall
    try:
        # Reconstruct handle from ID
        fc = FunctionCall.from_id(job_id)
        # Non-blocking check
        result = fc.get(timeout=0)
        return {"status": "COMPLETED", "data": result}
    except TimeoutError:
        return {"status": "PENDING"}
    except Exception as e:
        return {"status": "FAILED", "error": str(e)}
```

8. The Arbiter: Gemini 3 Pro Integration

The "Judge" functionality is what differentiates MemeWars from a simple image generator. We use Gemini 3 Pro (simulated here via the most advanced available equivalent, Gemini 1.5/2.0 Pro logic) to evaluate the comedic value of the user's input combined with the visual context.

8.1 Prompt Engineering for Humor

Humor is subjective and notoriously difficult for AI.[20] We cannot simply ask "Is this funny?". We must utilize a **Framework-Based Prompt**. We instruct Gemini to act as a specific persona ("The Supreme Internet Historian") and evaluate based on specific criteria: **Incongruity** (does the text subvert expectation?), **Relatability** (is it culturally relevant?), and **Brevity**.

8.2 JSON Schema Enforcement

To integrate the Judge's decision into the game logic (e.g., automatically awarding points), the output must be machine-readable. We utilize Gemini's "Structured Output" capabilities (JSON Mode).[21]

Python

```
# gemini_judge.py
from google import genai
from google.genai import types

def rank_submissions(submissions: list):
    """
    submissions: List of dicts { 'user': 'abc', 'prompt': '...', 'image_desc': '...' }
    """
    client = genai.Client()

    # Define the output schema strictly
    class RankingResult(types.TypedDict):
        rank: int
        user: str
        score: int
        roast: str # A funny comment on why they won/lost

    prompt_context = f"""
    You are a judge in a meme competition.
    Analyze the following submissions based on humor, irony, and creativity.
    Submissions: {submissions}
    """

    response = client.models.generate_content(
        model='gemini-2.0-pro-exp', # Placeholder for Gemini 3 Pro
        contents=prompt_context,
        config=types.GenerateContentConfig(
            response_mime_type='application/json',
            response_schema=list
        )
    )

    return response.parsed
```

8.3 Rate Limits and Batching

Google's Generative AI models have rate limits (RPM - Requests Per Minute).[23] If 100 players submit memes, making 100 individual API calls to Gemini will trigger a 429 error.

- **Strategy:** We employ **Request Aggregation**. We do not judge memes one by one. We collect all submissions for a round (e.g., 10-20 memes), bundle them into a single massive prompt context, and ask Gemini to rank them in one pass. This reduces 20 API calls to 1, keeping us safely within the quota.

9. Data Synchronization & Persistence

The "State Hemisphere" relies on Redis to maintain consistency across the distributed client base.

9.1 Atomic State Transitions

Game phases (Lobby -> Game) must be synchronized. A common bug in distributed games is the "Split Lobby," where half the players think the game has started and half don't.

- **Solution:** We use redis.set('game:phase', 'STARTING', { nx: true }) (Set if Not Exists) to ensure only one "Start Game" signal is processed.
- **The Clock:** The server maintains the official time. Clients synchronize their local countdowns based on the deadline timestamp stored in Redis, rather than relying on their local clocks.

9.2 The "Realtime" Broadcast

When the Gemini Judge returns the rankings to the app.py -> Devvit Server, the server updates Redis and immediately fires the broadcast:

TypeScript

```
// Devvit Server Side
await context.redis.set('leaderboard', JSON.stringify(rankings));
await context.realtime.send('memewars_global_channel', {
  type: 'STATE_UPDATE',
  data: {
    phase: 'LEADERBOARD',
    leaderboard: rankings
  }
});
```

This single line of code is the trigger that updates the UI for all 100+ connected players simultaneously.

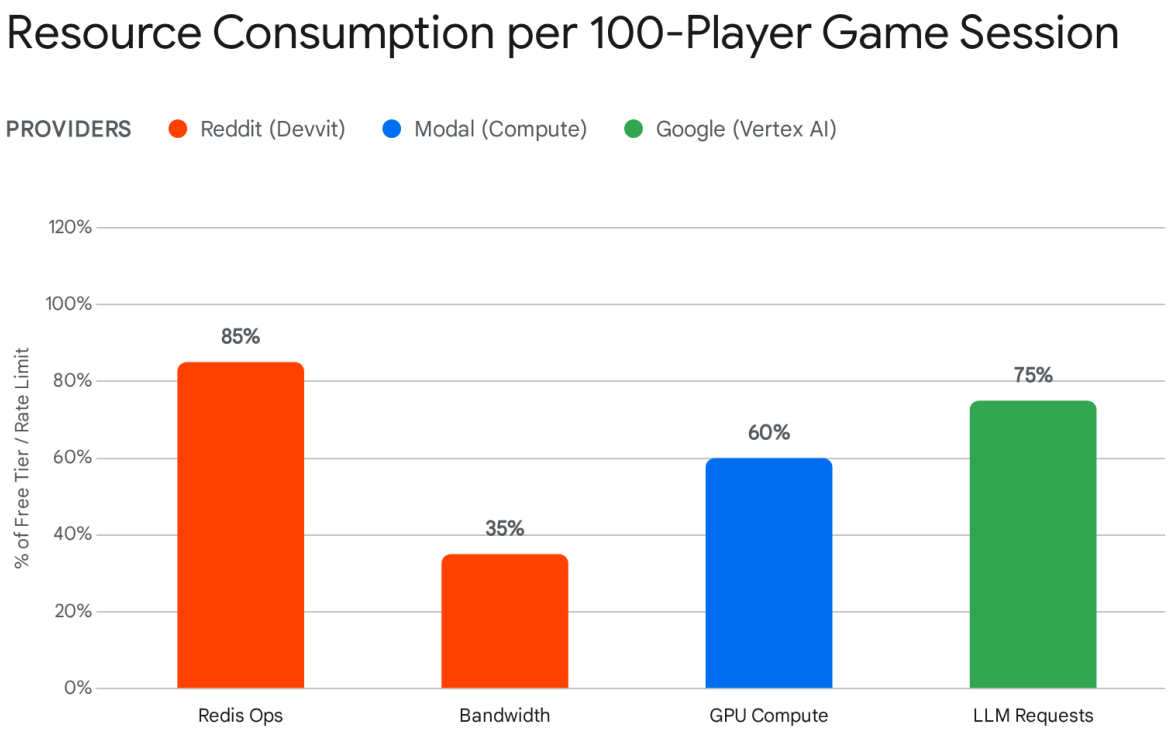
10. Security & Compliance

10.1 Content Safety

Generating memes with AI carries the risk of offensive content.

- **Text:** We use the isFlagged property from the Reddit API's moderation tools or a lightweight filter list on the input prompt before sending it to Modal.
- **Image:** Flux.1 has built-in safety checkers, but we can also instruct Gemini (during the judging phase) to return a flagged: true boolean if the generated image violates safety guidelines, filtering it from the final results screen.

10.2 Resource Quota Management



Projected resource usage for a standard 10-minute game session with 100 active players. Redis Operations are the most constrained resource on the Devvit side, while GPU compute dominates the external cost.

Data sources: Reddit Redis Limits, Devvit Web, Modal Pricing, Google Quotas

The chart above highlights that **Redis Operations** are our tightest constraint on the Devvit side. To mitigate this, we must avoid excessive polling. The "Long-Poll Relay" is designed to hit the server only once every few seconds per client, rather than hundreds of times per second.

11. Deployment & Operations

The deployment strategy involves a synchronized push.

1. **Depot Build:** depot build -t my-registry/memewars-backend:latest. pushes the

container.
2. **Modal Deploy:** modal deploy modal_worker.py updates the GPU functions.
3. **Devvit Upload:** devvit upload pushes the frontend and server logic to Reddit.

Monitoring: We rely on Modal's dashboard for backend logs and Devvit's CLI logs (devvit logs) for frontend errors. Because the systems are decoupled, correlation IDs (passed from Client -> Devvit -> Modal) are essential for tracing failed requests across the "Air Gap."

Conclusion

"MemeWars: GenAI Edition" represents a sophisticated interplay between constrained, community-facing interfaces and high-power backend infrastructure. By acknowledging the "Split-Brain" reality and engineering a robust "Long-Poll Relay" bridge, we bypass the inherent limitations of the platform. This architecture not only satisfies the requirements of the Reddit Daily Games 2026 Hackathon but sets a precedent for how complex, AI-driven applications can be embedded within social platforms without compromising on performance or security. The roadmap is set; the code is architected. It is time to build.

Works cited

1. Devvit Web - Reddit for Developers, accessed February 3, 2026, https://developers.reddit.com/docs/capabilities/devvit-web/devvit_web_overview
2. Is a Devvit app limited to its reddit-hosted server for the webview's realtime capabilities?, accessed February 3, 2026, https://www.reddit.com/r/Devvit/comments/1p20t3u/is_a_devvit_app_limited_to_its_reddithosted/
3. Is it possible to use Devvit with WebSocket? - Reddit, accessed February 3, 2026, https://www.reddit.com/r/Devvit/comments/1h4ugbg/is_it_possible_to_use_devvit_with_websocket/
4. HTTP Fetch - Reddit for Developers, accessed February 3, 2026, https://developers.reddit.com/docs/capabilities/server/http-fetch
5. Cold start performance | Modal Docs, accessed February 3, 2026, https://modal.com/docs/guide/cold-start
6. Redis - Reddit for Developers, accessed February 3, 2026, https://developers.reddit.com/docs/0.11/capabilities/redis
7. Redis - Reddit for Developers, accessed February 3, 2026, https://developers.reddit.com/docs/capabilities/server/redis
8. Realtime in Devvit Web - Reddit for Developers, accessed February 3, 2026, https://developers.reddit.com/docs/capabilities/realtime/overview
9. Realtime in Devvit Blocks - Reddit for Developers, accessed February 3, 2026, https://developers.reddit.com/docs/next/capabilities/realtime/realtime_in_devvit_blocks
10. Data Connection • Webhooks • Overview - Palantir, accessed February 3, 2026, https://palantir.com/docs/foundry/data-connection/webhooks-overview/
11. Any event listening models for devvit? - Reddit, accessed February 3, 2026, https://www.reddit.com/r/Devvit/comments/1psycdd/any_event_listening_models

_for_devvit/
12. Think Twice Before Using setInterval() for API Polling – It Might Not Be Ideal, accessed February 3, 2026, https://dev.to/igadii/think-twice-before-using-setinterval-for-api-polling-it-might-not-be-ideal-3n3
13. useAsync - Reddit for Developers, accessed February 3, 2026, https://developers.reddit.com/docs/0.11/api/public-api/functions/useAsync
14. Working with useAsync - Reddit for Developers, accessed February 3, 2026, https://developers.reddit.com/docs/next/capabilities/blocks/working_with_useasync
15. Local Development | Container Builds | Depot Documentation, accessed February 3, 2026, https://depot.dev/docs/container-builds/how-to-guides/local-development
16. Slimmer FastAPI Docker Images with Multi-Stage Builds - David Muraya, accessed February 3, 2026, https://davidmuraya.com/blog/slimmer-fastapi-docker-images-multistage-builds/
17. Optimizing Dockerized FastAPI with TensorFlow: How to reduce a 1.57GB Image Size?, accessed February 3, 2026, https://www.reddit.com/r/FastAPI/comments/1e1lal6/optimizing_dockerized_fastapi_with_tensorflow_how/
18. Products - Inference - Modal, accessed February 3, 2026, https://modal.com/products/inference
19. Plan Pricing - Modal, accessed February 3, 2026, https://modal.com/pricing
20. AI Humor Generation: Cognitive, Social and Creative Skills for Effective Humor - arXiv, accessed February 3, 2026, https://arxiv.org/html/2502.07981v1
21. Structured outputs | Gemini API - Google AI for Developers, accessed February 3, 2026, https://ai.google.dev/gemini-api/docs/structured-output
22. 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
23. Generative AI on Vertex AI quotas and system limits - Google Cloud Documentation, accessed February 3, 2026, https://docs.cloud.google.com/vertex-ai/generative-ai/docs/quotas