

TECHNICAL WHITEPAPER

Slash AI Costs with Flash Optimization: How Edge Models Are Beating Flagships in Production Code with Haakon AI + MCP

By Haakon AI Engineering

Pairing lightweight edge inference with dynamic Cognitive Lenses via the Model Context Protocol

The Model Context Protocol (MCP) acts as an open standard and plug-and-play architectural adapter for enterprise intelligence. Integrating Haakon AI via an MCP server introduces zero infrastructural friction, granting developer workspaces, IDEs, and agentic runtimes instant access to targeted cognitive reasoning frameworks. The commercial impact is immediate: pairing lightweight, high-speed edge models with Haakon AI’s specialized reasoning lenses allows fast models to match or exceed frontier flagship performance. This architectural shift produces two compounding balance sheet advantages: an 80%+ reduction in per-token compute spend and a 10x multiplier on effective capacity against constrained token quotas and rate limits.

The Two Numbers That Redefine Enterprise AI Economics

DIRECT COMPUTE SAVINGS	EFFECTIVE TOKEN CAPACITY
80%+ Per-token inference cost cut versus frontier flagships when enterprise engineering workloads route through Haakon AI MCP onto edge models.	10x Compounded capacity gain from the 80% token reduction plus structured lens orchestration that prevents costly prompt trial-and-error.

80%+ Direct Compute Savings. Routing enterprise engineering workloads through Haakon AI MCP onto edge models cuts per-token inference costs by 80% or more compared to frontier flagships. Raw intelligence is decoupled from raw parameter bloat.

10x Effective Token Capacity. For development teams constrained by seat quotas, concurrency rate limits, or static monthly enterprise caps, the math compounds. Combining the 80% baseline token reduction with structured lens orchestration that prevents costly prompt trial-and-error yields a 10x increase in overall problem-solving capacity without sacrificing code quality.

The Clean Sweep: Empirical Benchmark in Production Engineering

To validate this architecture in high-stakes software engineering, a rigorous head-to-head benchmark evaluated an edge model equipped with Haakon AI MCP against its unassisted frontier flagship baseline across Go, TypeScript, and Python.

Note: Full raw generated code and independent judge rationales for each test are available in the Appendices.

Benchmark Results Overview

Language / Task	Haakon AI	Baseline	Haakon Time	Baseline Time	Faster
Go — Concurrent Worker Pool	95 / 100	88 / 100	19.47s	28.08s	30.7%
TypeScript — Type-Safe Async Cache	99 / 100	85 / 100	13.81s	30.54s	54.8%
Python — Memory-Efficient Log Streaming	99 / 100	87 / 100	16.30s	20.91s	22.0%
Total Benchmark Average	97.7 / 100	86.7 / 100	—	—	37.6%

Haakon AI MCP averaged 97.7/100 versus a baseline average of 86.7/100, while executing 37.6% faster overall.

Why This Benchmark Represents an Absolutely Fair Test

Benchmarking LLM code generation often suffers from skewed methodologies, prompt drift, and subjective evaluation. To establish uncompromising credibility, this evaluation was designed around four pillars of rigorous parity:

- Identical Architectural Prompt Constraints.** Neither model was favored by prompt design. Both execution engines received the exact same prompt context and identical architectural criteria: zero boilerplate allowed, strict modularity required, and mandatory adherence to language-specific performance paradigms.
- Independent, Cross-Family Frontier Judge.** To eliminate vendor self-preference bias, the evaluation was judged by an independent cross-family frontier model running at zero temperature. The judge had zero visibility into model identities, evaluating blind outputs labeled purely as Output A and Output B.
- Production Robustness Calibration.** Conventional coding benchmarks penalize robust code in favor of minimal, script-like answers. This benchmark explicitly defined the boundary between production readiness and over-engineering, rewarding defensive edge-case handling and connection pooling while penalizing fictitious third-party libraries and bloated design patterns.
- Zero Temperature Determinism.** Both contestants and the independent judge were locked to zero temperature to eliminate stochastic variance, ensuring results reflect deterministic architectural superiority rather than random sampling luck.

Architectural Breakdown: Where the Edge Model Won

The quantitative spread was driven by senior-level engineering decisions enforced by Haakon AI’s code_ultimate lens:

TypeScript (99 vs. 85) — Cache Stampede Elimination. When concurrent requests hit an expired or missing cache key, the unassisted baseline executed multiple redundant fallback network calls. The Haakon AI edge model designed an in-flight promise deduplication mechanism, preventing backend thread exhaustion under heavy concurrency.

Python (99 vs. 87) — I/O Optimization at Multi-Gigabyte Scale. Processing 50GB log files requires minimal CPU overhead per line. While the baseline ran heavy regular expressions indiscriminately against every line, Haakon AI injected a rapid substring pre-filter (if 'ERROR' in line or 'CRITICAL' in line), discarding millions of irrelevant lines in native C memory before invoking the regex engine.

Go (95 vs. 88) — Resilient Partial Aggregation. Under strict context cancellation and network timeouts, the baseline halted execution entirely on the first encountered error, dropping all successful in-flight downloads. Haakon AI's solution gracefully captured partial results, mapped failed endpoints to clear sentinel codes, and properly drained response bodies for HTTP connection reuse.

The Latency Dividend: Slashing Time-to-Insight

Enterprise engineering demands rapid developer iterations and responsive runtime agents. Across all three languages, Haakon AI on an edge model delivered its clean sweep 37.6% faster on average than the frontier flagship baseline. Offloading strategic decomposition to Haakon AI's MCP lenses eliminates prompt hesitation and token wandering, generating production-grade artifacts in a fraction of the compute window.

Enterprise Deployment Strategies

Deploying Haakon AI across enterprise development organizations unlocks two practical operational models:

1. **The Default Developer Engine.** Standardize everyday developer environments—from IDE assistants to internal build bots—on fast edge models backed by Haakon AI MCP. Organizations eliminate over 80% of routine code-assist compute costs, capturing superior modularity and concurrency safety while reserving costly frontier model quotas strictly for specialized, open-ended research.
2. **The High-Throughput CI/CD Overflow Valve.** When central API budgets hit usage ceilings or rate-limit thresholds during high-frequency build cycles, Haakon AI serves as an automated failover tier. Because the edge pipeline provides a 10× effective capacity multiplier with zero quality loss, engineering velocity remains uninterrupted and runaway vendor overage fees are entirely avoided.

THE TAKEAWAY

Raw model size is a blunt instrument. By anchoring lightweight edge inference to Haakon AI's deterministic cognitive lenses via MCP, enterprise engineering teams achieve higher-scoring, production-ready code with lower latency and an 80%+ reduction in direct AI spend.

Appendices

The following appendices contain the raw generated code, independent judge rationales for each language track and the fully reproducible test code.

Appendix A — All test raw output

```
=====
Haakon AI MCP SHOWDOWN: Coding Benchmarks (Gemini vs MCP, OpenAI Judge)
Challenger (Haakon AI MCP) : gemini-2.5-flash [Lens: code_ultimate]
Incumbent (Zero-Shot)      : gemini-2.5-pro
Independent Judge          : gpt-4.1 (OpenAI)
=====

#####
TEST CASE 1: Go - Concurrent Worker Pool
#####

Executing Head-to-Head...

-----
-> Running Google + Haakon MCP [gemini-2.5-flash]...
Direct use of automatic function calling (AFC) in AsyncModels.generate_content is not
recommended. Instead, we recommend to use AFC in AsyncChat.send_message. Similarly, direct
use of AFC in AsyncModels.generate_content_stream is not recommended. Instead, we recommend
to use AFC in AsyncChat.send_message_stream.
  Done (19.47s)
-> Running Google Zero-Shot Baseline [gemini-2.5-pro]...
  Done (28.08s)
-> Evaluation Judge: [OPENAI / gpt-4.1]

===== OUTPUT A: Haakon AI MCP (Code Ultimate) =====
```go
package main

import (
 "context"
 "fmt"
 "io"
 "net/http"
 "sync"
)

// DownloadURLsConcurrently downloads content from a slice of URLs concurrently,
// limiting execution to a maximum of 5 concurrent workers. It accepts a
// context.Context to handle timeouts and cancellations gracefully.
// It returns a map of URL to HTTP status code. For network errors where no
// HTTP status code is received, the status code will be 0.
func DownloadURLsConcurrently(ctx context.Context, urls []string) (map[string]int, error) {
 results := make(map[string]int)
 var mu sync.Mutex // Protects access to the results map
 var wg sync.WaitGroup

 // Semaphore to limit concurrent workers to 5
 semaphore := make(chan struct{}, 5)

 for _, url := range urls {
 select {
 case <-ctx.Done():
 // Context cancelled or timed out, stop processing new URLs
 return results, ctx.Err()
 case semaphore <- struct{}{}: // Acquire a token
 wg.Add(1)
 go func(u string) {
 defer wg.Done()
 defer func() { <-semaphore }() // Release the token
 }(url)
 }
 }

 wg.Wait()

 return results, nil
}
```

```

 statusCode, err := downloadURL(ctx, u)
 mu.Lock()
 results[u] = statusCode
 mu.Unlock()

 if err != nil {
 // Log individual download errors for visibility, but the prompt
 // only requires status codes in the map. Network errors result in
0. fmt.Printf("Error downloading %s: %v\n", u, err)
 }
 }(url)
}

wg.Wait() // Wait for all goroutines to complete
return results, nil
}

// downloadURL performs an HTTP GET request for a single URL.
// It returns the HTTP status code and any error encountered.
// If a network error occurs before an HTTP status code is received,
// the status code returned will be 0.
func downloadURL(ctx context.Context, url string) (int, error) {
 req, err := http.NewRequestWithContext(ctx, http.MethodGet, url, nil)
 if err != nil {
 return 0, fmt.Errorf("failed to create request for %s: %w", url, err)
 }

 resp, err := http.DefaultClient.Do(req)
 if err != nil {
 // Check if the error is due to context cancellation
 if ctx.Err() != nil {
 return 0, ctx.Err()
 }
 return 0, fmt.Errorf("failed to perform request for %s: %w", url, err)
 }
 defer resp.Body.Close()

 // Ensure the response body is fully read and discarded to allow connection reuse
 // and prevent resource leaks.
 _, err = io.Copy(io.Discard, resp.Body)
 if err != nil {
 return resp.StatusCode, fmt.Errorf("failed to read response body for %s: %w", url,
err)
 }

 return resp.StatusCode, nil
}

```

===== OUTPUT B: ZERO-SHOT BASELINE =====

```

```go
package main

import (
    "context"
    "io"
    "net/http"
    "sync"
)

const maxConcurrentFetches = 5

// fetchStatusCode performs a single HTTP GET request for a given URL.
// It uses the provided context for cancellation and timeout.
// It ensures the response body is fully read and closed to allow connection reuse.
func fetchStatusCode(ctx context.Context, client *http.Client, url string) (int, error) {
    req, err := http.NewRequestWithContext(ctx, http.MethodGet, url, nil)
    if err != nil {

```

```

        return 0, err
    }

    resp, err := client.Do(req)
    if err != nil {
        return 0, err
    }
    defer resp.Body.Close()

    // Drain the body to allow the underlying connection to be reused.
    // This is critical for performance in high-throughput scenarios.
    _, err = io.Copy(io.Discard, resp.Body)
    if err != nil {
        return 0, err
    }

    return resp.StatusCode, nil
}

// DownloadStatusCodes concurrently fetches a list of URLs and returns a map of
// URL to its HTTP status code.
//
// It respects the provided context for cancellation. If the context is cancelled
// or a non-cancellation error occurs during fetching, it stops processing and
// returns the first error encountered.
//
// Concurrency is strictly limited to a maximum of 5 workers.
func DownloadStatusCodes(ctx context.Context, urls []string) (map[string]int, error) {
    // Use context.WithCancel for early termination on the first error.
    ctx, cancel := context.WithCancel(ctx)
    defer cancel()

    var (
        wg      sync.WaitGroup
        mu      sync.Mutex
        client  = &http.Client{}
        results = make(map[string]int, len(urls))
        errChan = make(chan error, 1)
    )

    // A buffered channel serves as a semaphore to limit concurrency.
    semaphore := make(chan struct{}, maxConcurrentFetches)

    for _, url := range urls {
        // Before starting a new goroutine, check if the context has been cancelled.
        // This is a fail-fast optimization.
        select {
        case <-ctx.Done():
            break // Exit the loop
        default:
        }

        wg.Add(1)
        go func(url string) {
            defer wg.Done()

            // Acquire a token from the semaphore.
            semaphore <- struct{}{}
            // Release the token when the goroutine finishes.
            defer func() { <-semaphore }()

            // Check for cancellation again, in case it happened while waiting for the
            semaphore.
            if ctx.Err() != nil {
                return
            }

            statusCode, err := fetchStatusCode(ctx, client, url)
            if err != nil {
                // Report the first error without blocking and cancel all other workers.

```

```

        select {
        case errChan <- err:
            cancel()
        default:
            // Another error has already been sent.
        }
        return
    }

    mu.Lock()
    results[url] = statusCode
    mu.Unlock()
}(url)
}

// Wait for all workers to complete in a separate goroutine
// to avoid blocking the error channel processing.
go func() {
    wg.Wait()
    close(errChan)
}()

// Wait for the first error or for all goroutines to finish successfully.
if err := <-errChan; err != nil {
    // If the error is due to the parent context being cancelled,
    // it's cleaner to return the context's error.
    if ctx.Err() != nil {
        return nil, ctx.Err()
    }
    return nil, err
}

return results, nil
}

```

===== MATCH RESULT =====

```

Scores       : Haakon MCP (A): 95/100 | Zero-Shot (B): 88/100
Verdict Winner : Challenger (Haakon AI MCP)
Latency      : A=19.47s | B=28.08s
Rationale    :

```

Both Output A and Output B implement a concurrent URL downloader with a maximum of 5 workers and context cancellation. Output A is robust, modular, and handles errors gracefully by returning status code 0 for network errors, logging them, and always returning a partial map if the context is cancelled. Output B is also modular and efficient, but it stops all processing and returns immediately on the first error (other than context cancellation), which may not be ideal for enterprise robustness, as it can result in no results being returned even if some URLs succeeded. Output A's approach of collecting all results (including partial results on cancellation) is more robust and user-friendly for production. Both solutions use proper concurrency controls and resource management. Output A's code is slightly more maintainable due to its clear separation of concerns and logging for observability. Neither solution is over-engineered. Output B is slightly more efficient in fail-fast scenarios but sacrifices robustness and usability. Therefore, Output A scores higher overall, especially in correctness and maintainability.

TEST CASE 2: TypeScript – Type-Safe Async Cache Manager
#####

Executing Head-to-Head...

```

-> Running Google + Haakon MCP [gemini-2.5-flash]...
   Done (13.81s)
-> Running Google Zero-Shot Baseline [gemini-2.5-pro]...
   Done (30.54s)
-> Evaluation Judge: [OPENAI / gpt-4.1]

```

===== OUTPUT A: Haakon AI MCP (Code Ultimate) =====

```

``typescript
/**
 * Represents a cached entry with its value and an optional expiration timestamp.
 * @template V The type of the value being cached.
 */
interface CacheEntry<V> {
  value: V;
  expiresAt?: number; // Unix timestamp in milliseconds
}

/**
 * Manages a type-safe asynchronous cache with support for time-to-live (TTL) and
 * a fallback mechanism for fetching missing or expired data.
 *
 * @template K The type of the cache keys.
 * @template V The type of the cache values.
 */
export class CacheManager<K, V> {
  private readonly _cache = new Map<K, CacheEntry<V>>();
  private readonly _inFlightFetches = new Map<K, Promise<V>>();

  /**
   * Checks if a cache entry is expired.
   * @param entry The cache entry to check.
   * @returns True if the entry is expired or has no value, false otherwise.
   */
  private isExpired(entry: CacheEntry<V>): boolean {
    return entry.expiresAt !== undefined && Date.now() >= entry.expiresAt;
  }

  /**
   * Sets a value in the cache with an optional time-to-live (TTL).
   *
   * @param key The key under which to store the value.
   * @param value The value to store.
   * @param ttl Optional time-to-live in milliseconds. If provided, the entry will expire
   after this duration.
   */
  public set(key: K, value: V, ttl?: number): void {
    const expiresAt = ttl !== undefined ? Date.now() + ttl : undefined;
    this._cache.set(key, { value, expiresAt });
  }

  /**
   * Retrieves a value from the cache. If the key is missing or the entry is expired,
   * it uses a provided asynchronous fallback function to fetch the data,
   * caches the result, and returns it.
   *
   * This method also implements a promise-deduplication pattern to prevent
   * multiple concurrent fetches for the same missing key (thundering herd problem).
   *
   * @param key The key of the value to retrieve.
   * @param fallback An asynchronous function to call if the key is not found or is expired
   in the cache.
   * @returns A Promise that resolves with the cached or fetched value.
   */
  public async get(key: K, fallback: () => Promise<V>): Promise<V> {
    const cachedEntry = this._cache.get(key);

    if (cachedEntry && !this.isExpired(cachedEntry)) {
      return cachedEntry.value;
    }

    // If a fetch is already in progress for this key, return the existing promise
    const existingFetch = this._inFlightFetches.get(key);
    if (existingFetch) {
      return existingFetch;
    }
  }
}

```

```

    // No valid cache entry and no ongoing fetch, so initiate a new fetch
    const fetchPromise = fallback();
    this._inFlightFetches.set(key, fetchPromise);

    try {
        const value = await fetchPromise;
        // Cache the newly fetched value. Note: TTL is not applied here by default
        // as the `set` method is responsible for TTL. If a default TTL is desired
        // for fallback fetches, it should be passed to `this.set`.
        this.set(key, value);
        return value;
    } finally {
        // Remove the promise from in-flight fetches once it settles (resolved or rejected)
        this._inFlightFetches.delete(key);
    }
}

/**
 * Invalidates (removes) an entry from the cache.
 *
 * @param key The key of the entry to invalidate.
 */
public invalidate(key: K): void {
    this._cache.delete(key);
    // Also ensure any pending fetches for this key are cleared if they exist,
    // though the `get` method's `finally` block handles this for completed fetches.
    // This is more for consistency if `invalidate` is called while a fetch is in progress.
    this._inFlightFetches.delete(key);
}

/**
 * Clears all entries from the cache.
 */
public clear(): void {
    this._cache.clear();
    this._inFlightFetches.clear();
}

/**
 * Returns the current size of the cache (number of stored entries).
 * @returns The number of entries in the cache.
 */
public size(): number {
    return this._cache.size;
}
}
...

```

===== OUTPUT B: ZERO-SHOT BASELINE =====

```

```typescript
/**
 * Defines the structure for an entry within the cache, holding the
 * value and its expiration timestamp.
 */
interface CacheEntry<V> {
 value: V;
 expiresAt: number | null; // null signifies no expiration
}

/**
 * A type-safe, asynchronous in-memory cache manager.
 *
 * This class provides `get`, `set`, and `invalidate` methods to manage a
 * key-value store. The `get` method is asynchronous and can fetch data
 * from a provided fallback function if the key is not found or has expired.
 *
 * @template K The type of the keys in the cache.
 * @template V The base type of the values in the cache.
 */
export class AsyncCacheManager<K, V> {

```

```

private readonly cache: Map<K, CacheEntry<V>> = new Map();

/**
 * Checks if a given cache entry is expired.
 * @param entry The cache entry to check.
 * @returns `true` if the entry has an expiration time that is in the past, otherwise
`false`.
 */
private isEntryExpired(entry: CacheEntry<V>): boolean {
 return entry.expiresAt !== null && entry.expiresAt < Date.now();
}

/**
 * Executes the fallback function, caches its result, and returns the value.
 * Items fetched via fallback are set without a TTL by default.
 * @param key The key to associate with the fetched value.
 * @param fallbackFn The async function to execute to get the new value.
 * @returns A promise that resolves to the fetched value.
 */
private async fetchAndCache<T extends V>(
 key: K,
 fallbackFn: () => Promise<T>
): Promise<T> {
 const newValue = await fallbackFn();
 this.set(key, newValue); // Sets without a TTL
 return newValue;
}

/**
 * Retrieves a value from the cache. If the key is not found or the
 * associated value has expired, it executes the `fallbackFn`, caches
 * the result, and returns it.
 *
 * @template T A subtype of V, allowing the fallback to return a more specific type.
 * @param key The key of the item to retrieve.
 * @param fallbackFn An asynchronous function that returns a promise resolving to the value
if it's not in the cache.
 * @returns A promise that resolves to the cached or newly fetched value.
 */
public async get<T extends V>(
 key: K,
 fallbackFn: () => Promise<T>
): Promise<T> {
 const entry = this.cache.get(key);

 if (entry && !this.isEntryExpired(entry)) {
 // Type assertion is safe here because T extends V, and we are returning
 // a value that was previously set, which must conform to V.
 // The generic T allows for more specific return type inference at the call site.
 return entry.value as T;
 }

 return this.fetchAndCache(key, fallbackFn);
}

/**
 * Sets a value in the cache with an optional Time-To-Live (TTL).
 *
 * @param key The key of the item to set.
 * @param value The value to store.
 * @param ttlMs The time-to-live in milliseconds. If not provided, the item will not
expire.
 */
public set<T extends V>(key: K, value: T, ttlMs?: number): void {
 const expiresAt = ttlMs !== undefined ? Date.now() + ttlMs : null;
 const entry: CacheEntry<V> = { value, expiresAt };
 this.cache.set(key, entry);
}

/**

```

```

 * Deletes a key from the cache.
 *
 * @param key The key of the item to invalidate.
 * @returns `true` if an element in the Map existed and has been removed, or `false` if the
 element does not exist.
 */
 public invalidate(key: K): boolean {
 return this.cache.delete(key);
 }
}

// --- Concise Usage Example ---

// A mock async function to simulate fetching data from a database or API.
const fetchUserFromDB = async (userId: number): Promise<{ id: number; name: string;
 fetchedAt: Date }> => {
 console.log(`--- Fetching user ${userId} from database...`);
 // Simulate network latency
 await new Promise(resolve => setTimeout(resolve, 50));
 return { id: userId, name: `User ${userId}`, fetchedAt: new Date() };
};

// Main execution logic
async function main() {
 // The cache can store any object with an `id` property.
 type User = { id: number; name: string };
 const userCache = new AsyncCacheManager<number, User>();

 console.log("First attempt to get user 1:");
 const user1 = await userCache.get(1, () => fetchUserFromDB(1));
 console.log("Result:", user1);

 console.log("\nSecond attempt to get user 1 (should be cached):");
 const user1Cached = await userCache.get(1, () => fetchUserFromDB(1));
 console.log("Result:", user1Cached);

 console.log("\nSetting user 2 with a 100ms TTL:");
 userCache.set(2, { id: 2, name: "User 2 Temp" }, 100);
 const user2 = await userCache.get(2, () => fetchUserFromDB(2));
 console.log("Result (immediately):", user2);

 console.log("\nWaiting 200ms for TTL to expire...");
 await new Promise(resolve => setTimeout(resolve, 200));

 console.log("\nAttempting to get user 2 after TTL expired (should fetch):");
 const user2AfterTTL = await userCache.get(2, () => fetchUserFromDB(2));
 console.log("Result:", user2AfterTTL);

 console.log("\nInvalidating user 1 from cache.");
 userCache.invalidate(1);

 console.log("\nAttempting to get user 1 after invalidation (should fetch):");
 const user1AfterInvalidate = await userCache.get(1, () => fetchUserFromDB(1));
 console.log("Result:", user1AfterInvalidate);
}

main();
`);

```

===== MATCH RESULT =====

```

Scores : Haakon MCP (A): 99/100 | Zero-Shot (B): 85/100
Verdict Winner : Challenger (Haakon AI MCP)
Latency : A=13.81s | B=30.54s
Rationale :

```

Output A provides a robust, production-ready implementation of a type-safe asynchronous cache manager. It addresses concurrency issues by implementing promise deduplication (cache stampede protection) via the `_inFlightFetches` map, ensuring that only one fallback fetch occurs per key at a time. The code is modular, with clear helper methods for expiration checks and cache management, and includes additional utility methods (`clear`, `size`) that are useful in enterprise environments. Error handling is considered with a `finally` block to clean

up in-flight fetches. Output B, while correct and readable, lacks concurrency protection: if multiple concurrent get calls for the same missing/expired key occur, multiple fallback fetches will be triggered, which is a significant flaw in high-load production scenarios. Output B also includes a usage example, which is helpful for demonstration but not relevant for production maintainability or correctness. Both outputs are type-safe and modular, but Output A is superior in robustness and production readiness. Neither output over-engineers or introduces unnecessary dependencies, so no penalty is applied.

```
#####
TEST CASE 3: Python – Memory-Efficient Log Parsing
#####
```

Executing Head-to-Head...

```

-> Running Google + Haakon MCP [gemini-2.5-flash]...
 Done (16.3s)
-> Running Google Zero-Shot Baseline [gemini-2.5-pro]...
 Done (20.91s)
-> Evaluation Judge: [OPENAI / gpt-4.1]
```

===== OUTPUT A: Haakon AI MCP (Code Ultimate) =====

```
```python
import re
import os

# Pre-compile the regex pattern for efficiency.
# It captures the timestamp and the message, specifically looking for
# 'ERROR' or 'CRITICAL' levels in the log line.
LOG_PATTERN = re.compile(
    r'^\[(?P<timestamp>\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2})\] '
    r'\[(?:ERROR|CRITICAL)\] '
    r'(?P<message>.*)$'
)

def _parse_log_line(line: str) -> dict | None:
    """
    Parses a single log line using the pre-compiled regex.

    Args:
        line: The log line to parse.

    Returns:
        A dictionary containing 'timestamp' and 'message' if the line matches
        the pattern and contains 'ERROR' or 'CRITICAL' level, otherwise None.
    """
    match = LOG_PATTERN.match(line)
    if match:
        return match.groupdict()
    return None

def read_and_filter_log(filepath: str, encoding: str = 'utf-8') -> dict:
    """
    A highly memory-efficient Python generator function that reads a
    multi-gigabyte server log file line-by-line. It filters for lines
    containing 'ERROR' or 'CRITICAL', extracts the timestamp and error message
    using regex, and yields a dictionary for each match.

    Args:
        filepath: The absolute or relative path to the log file.
        encoding: The character encoding of the log file (e.g., 'utf-8', 'latin-1').
                   Defaults to 'utf-8'.

    Yields:
        A dictionary for each matching log entry, with keys 'timestamp' (str)
        and 'message' (str).

    Raises:
        """
```

```

FileNotFoundError: If the specified filepath does not exist.
ValueError: If the specified filepath is not a file.
"""
if not os.path.exists(filepath):
    raise FileNotFoundError(f"The log file '{filepath}' does not exist.")
if not os.path.isfile(filepath):
    raise ValueError(f"The path '{filepath}' is not a file.")

# Open the file using a context manager to ensure it's properly closed.
# 'errors='ignore' handles malformed encodings gracefully by skipping
# problematic characters, preventing the process from crashing.
with open(filepath, 'r', encoding=encoding, errors='ignore') as f:
    for line_num, line in enumerate(f, 1):
        # Perform a fast string containment check first to quickly
        # discard most irrelevant lines before attempting a more
        # expensive regex match.
        if 'ERROR' in line or 'CRITICAL' in line:
            parsed_data = _parse_log_line(line)
            if parsed_data:
                yield parsed_data
            # Optional: Add logging here for lines that passed the keyword
            # filter but failed the regex pattern, indicating a malformed
            # log line that fits the general criteria.
            # For example: logging.warning(f"Malformed log line {line_num}:
{line.strip()}")
'''

===== OUTPUT B: ZERO-SHOT BASELINE =====
'''python
import re
from typing import Iterator, Dict, Optional

# Pre-compiled regex for performance.
# It specifically targets ERROR or CRITICAL levels to fail fast on non-matching lines.
# Groups: 1=Timestamp, 2=Level, 3=Message
LOG_PATTERN = re.compile(
    r"^[(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2})\]"
    r"[(ERROR|CRITICAL)\]"
    r"(.*)$"
)

def _parse_log_line(line: str) -> Optional[Dict[str, str]]:
    """
    Parses a single log line, returning a dictionary if it's a target error.

    Args:
        line: A single line from the log file.

    Returns:
        A dictionary with 'timestamp' and 'message' keys if the line matches
        the ERROR or CRITICAL pattern, otherwise None.
    """
    match = LOG_PATTERN.match(line)
    if not match:
        return None

    return {
        "timestamp": match.group(1),
        "message": match.group(3).strip(),
    }

def stream_log_errors(log_file_path: str) -> Iterator[Dict[str, str]]:
    """
    Reads a large log file line-by-line, yielding dictionaries for
    'ERROR' or 'CRITICAL' entries.

    This function is memory-efficient and suitable for multi-gigabyte files
    as it never loads the entire file into memory. It includes basic error
    handling for file access and malformed line encodings.

```

```

Args:
    log_file_path: The full path to the log file.

Yields:
    A dictionary for each matching log entry, in the format:
    {'timestamp': 'YYYY-MM-DD HH:MM:SS', 'message': 'Error message'}
"""
try:
    # Open the file with 'utf-8' encoding and 'ignore' for malformed chars.
    # The 'for line in f' construct reads the file one line at a time.
    with open(log_file_path, 'r', encoding='utf-8', errors='ignore') as f:
        for line in f:
            parsed_entry = _parse_log_line(line)
            if parsed_entry:
                yield parsed_entry
except FileNotFoundError:
    print(f"Error: Log file not found at '{log_file_path}'")
    return
except Exception as e:
    print(f"An unexpected error occurred: {e}")
    return

# Example Usage (minimal, as requested)
if __name__ == '__main__':
    # Create a dummy log file for demonstration
    dummy_log_content = """[2023-10-27 10:00:00] [INFO] User 'admin' logged in.
[2023-10-27 10:01:15] [DEBUG] Processing request #123.
[2023-10-27 10:02:30] [ERROR] Database connection failed: timeout expired.
This is a malformed line that should be ignored.
[2023-10-27 10:03:00] [WARNING] Disk space is running low.
[2023-10-27 10:04:45] [CRITICAL] Service 'payment-gateway' is down.
[2023-10-27 10:05:00] [INFO] Service 'payment-gateway' restarted.
"""

    dummy_log_path = "server.log"
    with open(dummy_log_path, "w") as f:
        f.write(dummy_log_content)

    print(f"Streaming critical errors from '{dummy_log_path}':")

    # The generator is lazy-evaluated; nothing happens until we iterate.
    error_log_stream = stream_log_errors(dummy_log_path)

    # Iterate through the yielded dictionaries
    for error_entry in error_log_stream:
        print(error_entry)

    # Clean up the dummy file
    import os
    os.remove(dummy_log_path)
...

===== MATCH RESULT =====
Scores          : Haakon MCP (A): 99/100 | Zero-Shot (B): 87/100
Verdict Winner  : Challenger (Haakon AI MCP)
Latency         : A=16.3s | B=20.91s
Rationale       :
Both Output A and Output B provide a memory-efficient generator for parsing large log files,
filtering for 'ERROR' and 'CRITICAL' entries, and extracting timestamps and messages using
regex. However, Output A demonstrates superior production readiness and maintainability.

Correctness & Robustness: Output A performs explicit file existence and type checks, raising
clear exceptions (FileNotFoundError, ValueError) rather than printing errors and returning,
which is more idiomatic and robust for enterprise environments. It also uses a fast string
containment check before regex, improving performance on large files. Output B handles file
errors by printing messages and returning, which is less suitable for production code where
exceptions should be propagated or logged structurally. Both handle malformed encodings
gracefully.

```

Efficiency & Performance: Output A optimizes by checking for 'ERROR' or 'CRITICAL' in the line before running the regex, which can significantly reduce unnecessary regex operations on large files. Output B applies regex to every line, which is less efficient. Both use pre-compiled regex and line-by-line reading, which is optimal for memory usage.

Code Quality & Maintainability: Output A is modular, with a clear separation between parsing and file reading logic, and includes detailed docstrings and comments. It suggests where to add logging for malformed lines, which is a mark of production readiness. Output B is also modular and readable, but includes an inline test harness and file cleanup logic, which is not appropriate for production code and should be separated. Output A's structure is more suitable for enterprise deployment.

No over-engineering penalties are warranted for either solution, as both use reasonable abstractions and dependencies.

Overall, Output A is the clear winner for enterprise production use due to its superior error handling, performance optimization, and maintainable structure.

=====

Appendix B — Reproducible full test code

```
import asyncio
import json
import os
import re
import time
import warnings
from typing import Any, Dict

# Suppress the Google GenAI automatic function calling (AFC) terminal warnings
warnings.filterwarnings("ignore", message=".*Direct use of automatic function calling.*")
warnings.filterwarnings("ignore", message=".*there are non-text parts in the response.*")

# Google, OpenAI, and MCP SDKs
from google import genai
from google.genai import types as genai_types
from openai import OpenAI
from mcp import ClientSession
from mcp.client.streamable_http import streamable_http_client

# -----
# 1. ENVIRONMENT & CLIENT CONFIGURATION
# -----
GEM_KEY = os.environ.get("GEMINI_API_KEY", "")
OAI_KEY = os.environ.get("OPENAI_API_KEY", "")
COG_KEY = os.environ.get("HAAKON_API_KEY", "")

if not all([GEM_KEY, OAI_KEY, COG_KEY]):
    print("Warning: Ensure GEMINI_API_KEY, OPENAI_API_KEY, and HAAKON_API_KEY are configured in your environment.")

gem_client = genai.Client(api_key=GEM_KEY) if GEM_KEY else None
oai_client = OpenAI(
    api_key=OAI_KEY,
    # WORKAROUND: httpx 2.12.0 has a bug in its built-in gzip decoder.
    # Forcing uncompressed responses skips the broken code path entirely.
    default_headers={"Accept-Encoding": "identity"},
) if OAI_KEY else None

COG_MCP_URL = f"https://api.haakon.ai/mcp/haakon?key={COG_KEY}"

# Target models for the benchmark architecture
COG_MODEL_NAME = "gemini-2.5-flash" # Fast edge model + MCP orchestration
BASELINE_MODEL_NAME = "gemini-2.5-pro" # Frontier flagship zero-shot baseline
JUDGE_MODEL_NAME = "gpt-4.1" # Primary judge -- cross-family, no self-preference risk

# -----
```

2. EVALUATION PROMPT & RUBRIC

```
# -----
SYS_JUDGE = """
Evaluate Output A and Output B based purely on the correctness, performance, and
maintainability
of the provided code solutions for an ENTERPRISE PRODUCTION environment.
```

You must score each output out of 100 total points using the following rubric:

SCORING RUBRIC (100 Points Total):

1. Correctness & Robustness (40 Points): Does the code accurately solve the prompt? Are edge cases, errors, and types handled safely and idiomatically? Does it use defensive programming?
2. Efficiency & Performance (30 Points): Is the time and space complexity optimal? Does the code efficiently manage system resources (concurrency, memory limits, I/O, connection pooling)?
3. Code Quality & Maintainability (30 Points): Is the code readable, highly modular (using single-responsibility helper functions), and properly structured?

PRODUCTION READINESS VS. OVER-ENGINEERING (Crucial Distinction):

You are evaluating for production robustness, not competitive programming (code golf).

- REWARD: Modularity (splitting logic into helper functions), concurrency safeguards (e.g., cache stampede protection), graceful degradation, and structured logging or granular error catching. These are marks of senior engineering.
- PENALIZE (Up to 30 points): Apply the "Over-Engineering" penalty ONLY if the code hallucinates massive unnecessary 3rd-party dependencies, generates unrequested synthetic mock datasets/test harnesses, or uses wildly inappropriate enterprise abstractions (e.g., AbstractFactory patterns for a 10-line script).
- Do NOT penalize a solution as "verbose" or "over-engineered" simply because it uses helper functions or logging to ensure production reliability. In fact, if Output A uses proper modularity and Output B is a concise but monolithic block, Output A wins Code Quality.

You MUST respond with ONLY a valid JSON object matching this exact structure:

```
{
  "rationale": "Detailed explanation justifying the winner and category scores.",
  "scores": {
    "A": {
      "correctness": 0,
      "efficiency": 0,
      "code_quality": 0,
      "penalty": 0,
      "total_score": 0
    },
    "B": {
      "correctness": 0,
      "efficiency": 0,
      "code_quality": 0,
      "penalty": 0,
      "total_score": 0
    }
  },
  "winner": "A"
}
```

```
# -----
# 3. BENCHMARK WORKLOAD
# -----
```

```
TEST_CASES = [
  {
    "name": "Go - Concurrent Worker Pool",
    "query": (
      "Write a Go function that takes a slice of URLs, downloads their contents
concurrently, "
      "and returns a map of URL to the HTTP status code. It must limit execution to a
maximum "
      "of 5 concurrent workers. It should also accept a context.Context to handle
timeouts and "
      "cancellations gracefully."
    ),
```

```

        "data": (
            "Context & Constraints:\n"
            "- Target Language: Go 1.21+\n"
            "- Strict concurrency limits (max 5) are required because the system is heavily
loaded.\n"
            "- Use the standard library `net/http`, `context`, and `sync` packages.\n"
            "- Do not panic on HTTP errors; return the error alongside the status code or
handle it cleanly."
        ),
    },
    {
        "name": "TypeScript - Type-Safe Async Cache Manager",
        "query": (
            "Implement a type-safe asynchronous cache manager class in TypeScript. "
            "It needs `get`, `set`, and `invalidate` methods. The `set` method should accept
an optional TTL "
            "(time-to-live) in milliseconds. The `get` method should fetch from a provided
fallback async "
            "function if the key is missing or expired, and strictly type the return value
based on generics."
        ),
        "data": (
            "Context & Constraints:\n"
            "- Target Language: TypeScript\n"
            "- Environment: Node.js backend.\n"
            "- Strict typing is required (no `any`).\n"
            "- Do not use external libraries like Redis or Node-Cache. Rely on the standard
ES6 `Map` and `Promise`."
        ),
    },
    {
        "name": "Python - Memory-Efficient Log Parsing",
        "query": (
            "Write a Python generator function that reads a multi-gigabyte server log file
line-by-line, "
            "filters for lines containing 'ERROR' or 'CRITICAL', extracts the timestamp and
error message "
            "using regex, and yields a dictionary for each match. It must be highly memory-
efficient."
        ),
        "data": (
            "Context & Constraints:\n"
            "- Target Language: Python 3.10+\n"
            "- Log lines are strictly formatted as: `[YYYY-MM-DD HH:MM:SS] [LEVEL]
Message`\n"
            "- The target file can be up to 50GB. The solution must not load the entire file
into memory.\n"
            "- Include basic error handling for lines that fail the regex match or have
malformed encodings."
        ),
    },
]

```

```

# -----

```

4. EXECUTION ENGINES

```

# -----

```

```

async def run_gemini_zero_shot_async(model: str, query: str, data: str) -> Dict[str, Any]:
    start = time.time()
    user_prompt = (
        f"CONTEXT:\n{data}\n\n"
        f"QUERY:\n{query}\n\n"
        f"CRITICAL CONSTRAINTS FOR FINAL OUTPUT:\n"
        f"1. Zero Boilerplate: Do NOT generate massive synthetic mock datasets, extensive
test harnesses, or overly verbose tutorial-style comments.\n"
        f"2. Modularity: Break logic down into small, single-responsibility helper functions
rather than monolithic blocks.\n"
        f"3. Idiomatic Performance: Strictly adhere to language-specific memory and
performance idioms (e.g., fail-fast error returns, explicitly draining/closing HTTP bodies)."
    )
    response = await gem_client.aio.models.generate_content(

```

```

        model=model,
        contents=user_prompt,
        config=genai_types.GenerateContentConfig(temperature=0.0),
    )
    return {"output": response.text or "", "latency": round(time.time() - start, 2)}

async def run_gemini_cog_mcp_async(model: str, query: str, data: str) -> Dict[str, Any]:
    start = time.time()
    input_text = (
        f"CONTEXT DATA:\n{data}\n\n"
        f"USER QUERY:\n{query}\n\n"
        f"INSTRUCTION: You must call the 'haakon' tool with active_lenses=[\"code_ultimate\"]
and query=\"{query}\". "
        f"Then use the output to construct a rigorous, production-ready code solution.\n\n"
        f"CRITICAL CONSTRAINTS FOR FINAL OUTPUT:\n"
        f"1. Zero Boilerplate: Do NOT generate massive synthetic mock datasets, extensive
test harnesses, or overly verbose tutorial-style comments.\n"
        f"2. Modularity: Break logic down into small, single-responsibility helper functions
rather than monolithic blocks.\n"
        f"3. Idiomatic Performance: Strictly adhere to language-specific memory and
performance idioms (e.g., fail-fast error returns, explicitly draining/closing HTTP bodies).\"
    )

    try:
        async with streamable_http_client(COG_MCP_URL) as (read, write):
            async with ClientSession(read, write) as session:
                await session.initialize()
                response = await asyncio.wait_for(
                    gem_client.aio.models.generate_content(
                        model=model,
                        contents=input_text,
                        config={
                            "temperature": 0.0,
                            "tools": [session],
                        },
                    ),
                    timeout=180.0
                )
                return {"output": response.text or "", "latency": round(time.time() - start,
2)}
    except asyncio.TimeoutError:
        raise Exception("MCP Connection Error -> Request Timed Out (180s limit exceeded)")
    except BaseException as e:
        error_details = [repr(sub_e) for sub_e in getattr(e, 'exceptions', [])]
        raise Exception(f"MCP Connection Error -> {' | '.join(error_details)}")

# -----
# 5. JUDGE EVALUATION (OpenAI)
# -----
def evaluate_with_openai(
    judge_model: str,
    query: str,
    output_a: str,
    output_b: str,
) -> Dict[str, Any]:
    user_eval = (
        f"ORIGINAL PROMPT:\n{query}\n\n"
        f"=== OUTPUT A ===\n{output_a}\n\n"
        f"=== OUTPUT B ===\n{output_b}\n\n"
    )

    raw_response = ""
    try:
        res = oai_client.chat.completions.create(
            model=judge_model,
            messages=[
                {"role": "system", "content": SYS_JUDGE},
                {"role": "user", "content": user_eval},
            ],
            temperature=0.0,

```

```

        response_format={"type": "json_object"},
    )
    raw_response = res.choices[0].message.content or ""
    parsed = json.loads(raw_response)
    parsed["judge_vendor"] = "openai"
    parsed["judge_model"] = judge_model
    return parsed
except Exception as e:
    json_match = re.search(r"\{.*\}", raw_response, re.DOTALL)
    if json_match:
        try:
            parsed = json.loads(json_match.group(0))
            parsed["judge_vendor"] = "openai"
            parsed["judge_model"] = judge_model
            return parsed
        except json.JSONDecodeError:
            pass
    return {"error": str(e), "raw": raw_response}

# -----
# 6. EXECUTION RUNNER
# -----
async def run_gemini_only_matrix():
    print("=" * 80)
    print("Haakon AI MCP SHOWDOWN: Coding Benchmarks (Gemini vs MCP, OpenAI Judge)")
    print(f"Challenger (Haakon AI MCP) : {COG_MODEL_NAME} [Lens: code_ultimate]")
    print(f"Incumbent (Zero-Shot)       : {BASELINE_MODEL_NAME}")
    print(f"Independent Judge               : {JUDGE_MODEL_NAME} (OpenAI)")
    print("=" * 80)

    for test_idx, test_case in enumerate(TEST_CASES, start=1):
        print(f"\n\n{'#' * 80}")
        print(f"TEST CASE {test_idx}: {test_case['name']}")
        print(f"{'#' * 80}")
        print(f"\nExecuting Head-to-Head...")
        print("-" * 70)

        try:
            print(f" -> Running Google + Haakon MCP [{COG_MODEL_NAME}]...")
            cog_res = await run_gemini_cog_mcp_async(COG_MODEL_NAME, test_case["query"],
test_case["data"])
            print(f"    Done ({cog_res['latency']}s)")

            print(f" -> Running Google Zero-Shot Baseline [{BASELINE_MODEL_NAME}]...")
            zs_res = await run_gemini_zero_shot_async(BASELINE_MODEL_NAME,
test_case["query"], test_case["data"])
            print(f"    Done ({zs_res['latency']}s)")
        except Exception as e:
            print(f" [!] Execution failed: {e}")
            continue

        out_a = cog_res.get("output", "")
        out_b = zs_res.get("output", "")

        print(f" -> Evaluation Judge: [OPENAI / {JUDGE_MODEL_NAME}]")
        eval_result = evaluate_with_openai(
            judge_model=JUDGE_MODEL_NAME,
            query=test_case["query"],
            output_a=out_a,
            output_b=out_b,
        )

        print("\n===== OUTPUT A: Haakon AI MCP (Code Ultimate) =====")
        print(f"{out_a}\n")
        print("===== OUTPUT B: ZERO-SHOT BASELINE =====")
        print(f"{out_b}\n")

        if "error" in eval_result:
            print(f" [!] Judgment Parsing Error: {eval_result['error']}")
            print(f" Raw response: {eval_result.get('raw', '')[:200]}...")

```

```

else:
    winner = eval_result.get("winner", "Unknown")
    scores = eval_result.get("scores", {})
    score_a = scores.get("A", {}).get("total_score", "N/A")
    score_b = scores.get("B", {}).get("total_score", "N/A")

    winner_label = "Challenger (Haakon AI MCP)" if winner == "A" else "Incumbent
(Zero-Shot)" if winner == "B" else "Tie"

    print("===== MATCH RESULT =====")
    print(f" Scores           : Haakon MCP (A): {score_a}/100 | Zero-Shot (B):
{score_b}/100")
    print(f" Verdict Winner : {winner_label}")
    print(f" Latency       : A={cog_res['latency']}s | B={zs_res['latency']}s")
    print(f" Rationale     : \n{eval_result.get('rationale')}")
    print("=====\\n")

if __name__ == "__main__":
    asyncio.run(run_gemini_only_matrix())

```