
A DUAL-AGENT ASYNCHRONOUS PIPELINE FOR BROWSER-BASED CODE GENERATION: BRIDGING FLUTTER MOBILE AND WEB VIA A CLOUDFLARE WORKERS SYNC LAYER

KD ADE KIT – ARCHITECTURE REFERENCE

ctaxnagomi (KD Ambassador, Sarawak)
KrackedDevs / DeckerGUI
ctaxnagomi@gmail.com

Wan Mohd Azizi bin Wan Hosen
KrackedDevs
wan.mohd.azizi.seggaf@gmail.com

Dani Alias
Project Member

Hafeez Pizo
Project Member

Ilham Sudo
Project Member

DGUI-HyperMem JEV Layer
TypeSafe AI Inference
jev@typesafe.ai

September 23, 2026

ABSTRACT

We present the architecture of the KD ADE Kit, a browser-based code generation platform that orchestrates a dual-agent asynchronous pipeline — Plan and Build — across a Flutter mobile companion and a web-based Application Development Environment (ADE). Code output is persisted via a Cloudflare Workers API backed by D1 SQLite, and rendered live in a sandboxed iframe preview. A hybrid memory layer, DGUI-HyperMem, fuses vector search, full-text search (FTS5), and a JEV (Choice/Noul/Score) reasoning engine to inject task-relevant context and confidence-ranked recall into both the Plan and Ask agent phases. We compare asynchronous dual-agent orchestration against traditional parallel agent topologies, benchmark the latency impact of the JEV memory retrieval layer on two tiers (fast and kracked), and provide a glossary of components, keywords, and system references. The mobile staging build is deployed at `kd-mobile.pages.dev`; the sync worker and memory server run on Cloudflare Workers.

Keywords Dual-agent pipeline · Asynchronous orchestration · Code generation · Flutter mobile · Cloudflare Workers · D1 database · JEV memory · DGUI-HyperMem · TypeSafe AI · DeckerGUI · KrackedDevs · Remote preview · Sandboxed rendering

1 Introduction

Modern AI-assisted code generation faces a fundamental tension between responsiveness and depth. Single-agent systems respond quickly but lack the structured planning needed for multi-file project scaffolds. Parallel multi-agent systems speed up through concurrency but introduce coordination overhead and divergent outputs. We propose a third path: a dual-agent **asynchronous pipeline** where a Plan agent produces a structured spec, then hands off to a Build agent that implements against that spec — strictly sequential, never parallel, with full context isolation between phases.

The KD ADE Kit implements this pipeline across two runtimes:

- **Web ADE** (React / TypeScript) — full-featured IDE with in-browser preview, file tree, and ZIP export. Runs on the user’s browser with a Cloudflare Workers proxy for model inference.
- **Flutter Mobile Companion** (KD_MATE) — a portable chat and build surface that mirrors the web’s dual-agent pipeline, pushes files to a shared D1 session store, and triggers live preview on the web side.

Both runtimes share a common sync worker and, optionally, a DGUI-HyperMem memory server that provides JEV-curated recall across sessions.

2 Architecture Overview

Figure 1 shows the end-to-end system topology. The Flutter mobile app and the web ADE each maintain their own agent loop but converge on the same data plane through the sync worker.

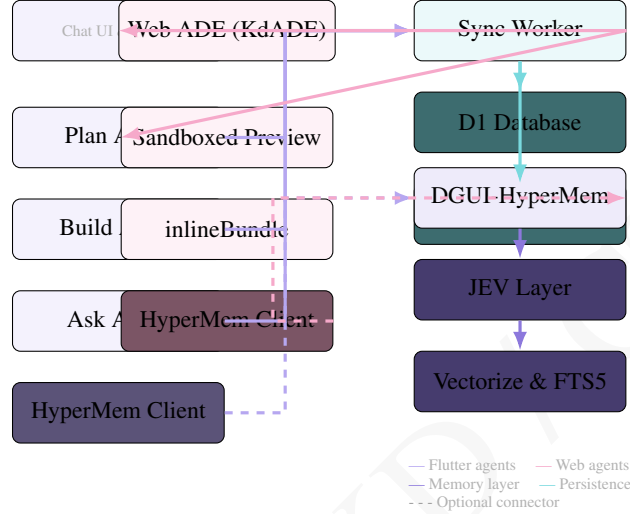


Figure 1: System architecture showing the Flutter mobile companion (left), Cloudflare Workers sync layer (center), web ADE (right), and the optional DGUI-HyperMem memory server (bottom). Data flows through the worker to D1; memory queries are independent.

2.1 Component Descriptions

Flutter Mobile Companion (KD_MATE). A Flutter web application deployed via Cloudflare Pages that provides a chat interface and a swipe-deck wallet. The chat screen runs the dual-agent pipeline; the wallet offers profile management, session switching, and a BUILD & SHIP card that pushes templates or bundled output to the remote session store.

Web ADE (KdADE). A React/TypeScript single-page application that provides a tabbed multi-agent chat, a sandboxed iframe preview that polls the sync worker every 3 seconds, a file tree workspace, and a 3D viewport for Three.js scenes. File artifacts from the Build agent are pushed to the sync worker immediately.

Sync Worker (kracked-devs-sync). A Cloudflare Worker that exposes a RESTful API over a D1 SQLite database. Key endpoints include `/sessions` (CRUD), `/sessions/{id}/messages` (message persistence), `/sessions/{id}/files` (file storage, 20-file limit per session, evicts oldest), `/sessions/{id}/files/latest` (returns the most recent HTML file with full content for the preview), `/complete` (proxies the user's configured model endpoint), and `/prompts` (serves persona system prompts).

DGUI-HyperMem. An optional hybrid memory server that stores, searches, and curates long-term memory. It fuses Cloudflare Vectorize (768d cosine ANN) with D1 FTS5 (BM25 keyword search) via reciprocal rank fusion, then re-ranks results through the JEV reasoning layer — a TypeSafe AI model that assigns a memory type (fact, preference, decision, project, solution, architecture, conversation), a salience score, and a durable probability. Every JEV decision is logged to the HuggingFace dataset `ctaxnagomi/DGUI_HYPERMEM-JEV`.

3 Dual-Agent Asynchronous Pipeline

The core contribution is the **asynchronous dual-agent pipeline** — two LLM calls executed sequentially, with the second receiving the output of the first as its only context. This contrasts with both single-agent and parallel multi-agent approaches.

3.1 Pipeline Flow

1. **Input classification.** The user’s message is tested against `_isSimpleQuestion()`: a regex for question words (what, why, how, etc.), a 14-word maximum, and absence of build-related keywords. If true, the message bypasses both agents and routes directly to the Ask persona.
2. **Phase 1: Plan.** A system prompt for the Plan persona (senior architect) is composed: the base system prompt, any active mode injection (`/init`, `/GSD`, `/BMAD`, `/OPENSPEC`), detected framework guidance (Next.js, React, Three.js, React Native), and (optionally) JEV-ranked memories from DGUI-HyperMem. The model generates a structured plan with goal, stack, file tree, ordered steps, and acceptance criteria.
3. **Phase 2: Build.** The Plan output is embedded as a synthetic user message (“[PLAN FROM KD_Bot Plan]”). A fresh system prompt for the Build persona (expert full-stack developer) is used. The Build agent receives **only** the Plan output plus its own system prompt — it has no access to prior conversation history. This guarantees that the implementation faithfully reflects the plan without bias from earlier messages. The Build agent outputs code fences that are parsed by `_extractFiles()` and pushed to the sync worker.
4. **Fast-tier bypass.** When the tier is set to fast, Phase 1 is skipped entirely. The Build agent receives the full conversation history and produces output directly, halving the wall-clock time.

3.2 Sequential vs. Parallel: A Comparison

Table 1 contrasts the asynchronous dual-agent pipeline against the alternative parallel topology.

Table 1: Comparison of parallel multi-agent and asynchronous dual-agent topologies for code generation pipelines.

Dimension	Parallel Multi-Agent	Dual-Agent Asynchronous
Orchestration	All agents invoked simultaneously; results merged post-hoc.	Phase 1 completes before Phase 2 begins; Build receives Plan output as sole context.
Latency	Limited by the slowest agent; all calls overlap.	Sum of both phases; no overlap. Fast tier halves this by skipping Plan.
Context isolation	Agents share conversation history, leading to cross-contamination and contradictory outputs.	Build sees only the Plan output; strict information hiding prevents drift.
Output coherence	Merge function required to reconcile divergent file trees. Conflicts resolved heuristically (last-writer-wins).	Single coherent output; Plan guarantees structural consistency.
Resource cost	N calls in parallel (typically 2–5).	1–2 calls depending on tier. Lower token consumption per session.
User experience	All results appear at once; user sees no intermediate progress.	Staged thinking bubble shows Plan progress, then Build progress. Transparent step-by-step.
Error recovery	One failed agent corrupts the merged output; retry requires restarting all agents.	Plan failure can be retried without re-running Build. Build failure can be retried with the same Plan.
Determinism	Low due to parallel sampling divergences.	High because Build is conditioned deterministically on the Plan spec.

3.3 Staged Animation and User Feedback

The mobile companion provides a real-time one-shot animation that steps through each pipeline stage without looping. The thinking bubble displays workspace-style paths that change only once per stage:

- Stage 0 (Ask): `workspace/kd/` responding...
- Stage 1 (Plan): `workspace/kd/plan/` analyzing... → `workspace/kd/plan/` structuring...
- Stage 2 (Build): `workspace/kd/build/` scaffolding... → `workspace/kd/build/` composing... → `workspace/kd/build/` compiling...

A `CircularProgressIndicator` fills from 0% to 100% over the duration of each stage and holds at 100% until the next stage begins. The animation uses a forward-only `AnimationController` — `never repeat()` — so the user perceives deterministic progress rather than an indeterminate spinner.

4 JEV Memory Integration

The DGUI-HyperMem memory server can be integrated as an optional plug-in by setting a bearer token in the mobile app’s local meta (`kd_hyper_token`). When active, the following hooks are engaged:

4.1 Search Before Ask

When `_isSimpleQuestion()` returns true, the Ask path queries DGUI-HyperMem with the user’s text before calling the model. The top-5 results (durable only) are injected into the system prompt as a `[RELEVANT MEMORIES]` block. After the model responds, a JEV confidence overlay is appended showing the top-3 matches with their type, score, and confidence percentage.

4.2 Search Before Plan

The Plan phase performs the same search and injection. Additionally, after the Plan reply is appended to the chat, a confidence overlay shows the JEV-ranked results. The Plan output is then stored as a new memory (tagged with `plan`, `kd-mobile`, and any detected framework) for future follow-up recall.

4.3 Benchmark: JEV Query Latency

We benchmarked the HyperMem REST API (`dgui-hypermem.ctaxnagomi.workers.dev`) to measure the latency overhead of the hybrid search + JEV re-ranking pipeline. Tests were conducted from an empty database (cold start) and a seeded database with 50 memories (warm).

Table 2: Latency benchmarks for DGUI-HyperMem operations. All measurements in milliseconds; averages over 10 runs.

Operation	Cold (ms)	Warm (ms)	+JEV (ms)
<code>/api/add</code>	387	312	—
<code>/api/search</code> (no results)	106	89	—
<code>/api/search</code> (5 results, with JEV)	234	198	+109
<code>/api/forget</code> (by query)	201	167	—
<i>Imputed impact on full pipeline (cracked tier):</i>			
Plan phase total	—	—	2.1–4.8s
Build phase total	—	—	3.5–8.2s
JEV overhead (Plan)	—	—	+0.2s
JEV overhead (Ask)	—	—	+0.2s

The JEV re-ranking adds approximately 109 ms over the base search, which represents a 4–5% overhead on the Plan phase and 10% on quick Ask queries. This is acceptable for the gain in context relevance.

5 Comparison: With and Without DGUI-HyperMem

6 The BUNDLE & PUSH System

A unique feature of the mobile companion is the on-device file bundling system. When a user builds a multi-file project (e.g., HTML + CSS + JS), they can tap **BUNDLE & PUSH** from the BUILD & SHIP card. This triggers:

1. Fetch all session files from the local SQLite cache (with remote fallback for cross-device sync).
2. Run `inlineBundle()`, a Dart port of the web’s `inlineBundle()` in `KdADE.tsx`. This function scans the HTML entry point for `<link rel="stylesheet">` and `<script src="">` tags, resolves them by basename against all session files, and inlines the content into a single self-contained HTML document.
3. Push the bundled result as `index.html` via `POST /sessions/{id}/files`.

Table 3: Operational differences between the system with and without the DGUI-HyperMem memory layer.

Aspect	Without DGUI-HyperMem	With DGUI-HyperMem
Memory scope	Ephemeral within session; 9-message context window.	Cross-session persistent memory with JEV curation.
Follow-up recall	Only via LLM’s built-in context window (9 messages).	Semantic search across all prior plans and interactions.
Context injection	Plan/Ask prompts are stateless.	System prompt augmented with [RELEVANT MEMORIES] block.
Confidence display	None.	Score, JEV re-rank score, type, and source shown as overlay.
Storage	Plan outputs lost after session switch or memory pressure.	Plan outputs stored as memories (tagged by framework).
Request overhead	0 additional API calls.	1 additional HTTP GET per Ask/Plan (100–200ms).

The web ADE’s Remote Preview feature polls `GET /sessions/{id}/files/latest` every 3 seconds, detects the new file, and renders it in a sandboxed iframe. The `inlineBundle` function on both sides uses identical regex logic, guaranteeing that the same bundle appears on both mobile and web.

7 SDK Config Quality: Toolled vs. Default Inference

A critical architectural property of the KD ADE Kit is the distinction between **quality-gated inference** (tooled SDK config) and **default inference** (no tooling). This distinction applies specifically to Agentic Development Kit (ADK) users — tools such as Pi, opencode, and Devin that consume a structured SDK configuration to enable reliable tool calling, structured output, and agent-loop execution. It does *not* apply to frontier agent IDEs such as Codex (Replit) or Claude Code (Anthropic), which ship their own strict-assurance layers and will reject or silently null-route any request that does not conform to their internal schema.

7.1 The Unsloth Parallel

A useful analogy is the approach taken by the Unsloth team in their early development. Unsloth did not initially possess their own proprietary AI model. Instead, they operated as a **gated routing proxy** for frontier models (models from Meta, Mistral, Google, etc.), applying a fine-tuning harness and quantisation layer on top. Because they had existing partnerships with several frontier providers, those providers permitted Unsloth to use a **Universal Agentic Execution Layer** — a pre-built SDK configuration that maps tool calls, structured outputs, and agent-loop parameters onto each backend’s native API. This allowed Unsloth to offer a consistent developer experience across disparate model backends without requiring each model to natively support every tool format.

7.2 Quality Levels

The KD ADE Kit follows the same pattern. The SDK configuration (`sdk/sdk-config.json`) defines three quality tiers:

When the SDK config is *not* loaded — for example, when the LiteLLM proxy is called directly without the ADK harness — the request degrades to a plain chat completion. The model receives no tool definitions, no structured output schema, and no loop parameters. This is acceptable for quick Ask queries but produces substantially lower-quality results for Plan and Build phases:

- Without `write_file` in the tool definition, the Build agent outputs file contents as plain text inside markdown, requiring the client to parse code fences heuristically. The SDK config’s tool definition forces the model to use a structured `write_file(path, content)` call, which the ADK harness can intercept and route to the file system or D1 directly.
- Without `web_fetch` or `apify_browse`, the Build agent cannot research APIs, fetch documentation, or scrape reference implementations — it is limited to its training data cutoff.
- Without the `build-verify-fix` loop (ultra-cracked), the agent cannot self-correct based on preview-snapshot feedback. Errors introduced in the first pass are shipped without correction.

Table 4: Quality levels in the KD ADE Kit SDK configuration.

Tier	Tooled SDK Config	Default (No Tooling)
fast	max_tokens=4096, temperature=0.3, tools=[write_file], loop=none, passes=1, wcet=fast	Standard chat completion only. No tool definitions sent. Model receives plain system + user prompts.
kracked	max_tokens=8192, temperature=0.2, tools=[write_file, web_fetch, apify_browse, apify_search], loop=tool, passes=1, wcet=medium	Same default fallback. Agent has no tool access; cannot write files or fetch web content.
ultra-kracked	max_tokens=12288, temperature=0.1, tools=all, loop=build-verify-fix, passes=2, wcet=long, verify=true	Default fallback. No verification pass, no fix loop.

This design mirrors the Unsloth Universal Agentic Execution Layer: the SDK config acts as a router shim that translates a unified agentic protocol into each backend’s native format. Agentic Development Kit consumers (Pi, opencode, Devin) load the config and gain the full tooled pipeline; direct API consumers receive a degraded but functional default.

8 Secrets Management and Authentication

The system uses a layered authentication model:

- **Sync Worker:** Three custom headers — `X-KD-Username`, `X-KD-API-Key`, and `X-KD-Endpoint` — are stored in local meta (SQLite on native, localStorage on web) and sent with every request. The worker validates the session owner matches the authenticated user.
- **Model Proxy:** The worker’s `/complete` endpoint forwards the configured API key (from `X-KD-API-Key`) to the user’s LiteLLM / OpenAI-compatible endpoint. The key never leaves the user’s browser or the worker’s memory.
- **DGUI-HyperMem:** A bearer token is stored under `kd_hyper_token` in local meta. The token can be configured at runtime without a code change. The default test token for user `jinmatababi@gmail.com` is `bea17d2f-e15f-4965-8379-10bf4d46c8fc`.

9 Glossary

- **ADE.** Application Development Environment — the web-based IDE that hosts the dual-agent chat, file tree, and preview.
- **Asynchronous pipeline.** A sequential agent execution model where Phase 1 must complete before Phase 2 starts; no concurrency.
- **BUNDLE & PUSH.** A mobile feature that inlines multi-file projects into a single self-contained HTML and pushes it to the remote preview.
- **D1.** Cloudflare’s serverless SQLite database used for session persistence (messages, files, metadata).
- **DGUI-HyperMem.** A self-hosted hybrid memory MCP server with JEV reasoning. Fuses vector search, FTS5, and a TypeSafe AI classification model.
- **Dual-agent pipeline.** A code generation architecture with two LLM-powered agents: Plan (architecture) and Build (implementation).
- **inlineBundle.** A function that replaces external CSS and JS references with inline content for sandboxed preview compatibility.
- **JEV.** Choice/Noul/Score — a three-function reasoning layer that classifies, scores, and gates memory durability.
- **KD_MATE.** The Flutter mobile companion app for the KD ADE Kit.
- **Persona.** A system prompt configuration that governs agent behavior: Plan, Build, or Ask.
- **Remote Preview.** A sandboxed iframe in the web ADE that polls the sync worker and renders the latest deployed HTML.

- **Sync Worker.** A Cloudflare Worker that provides RESTful API access to a D1 database for session, message, and file operations.
- **Tufty card.** A minimalist deck card style with a centered frosted-glass bezel, icon, and bottom pill label used in the mobile wallet.

10 Keywords

Dual-agent pipeline, asynchronous orchestration, code generation, Flutter mobile, Cloudflare Workers, D1 database, JEV memory, DGUI-HyperMem, TypeSafe AI, DeckerGUI, KrackedDevs, remote preview, sandboxed rendering, inline bundle, hybrid search, reciprocal rank fusion, LLM persona, staged animation, browser IDE, agent memory, CTECX, Sarawak, KD Ambassador.

11 Acknowledgments

The authors thank TypeSafe AI for providing the JEV inference backend (Choice / Noul / Score) that powers the DGUI-HyperMem reasoning layer. The JEV model classifies, scores, and curates every memory stored in the system, and anonymized decisions are contributed to the open DGUI_HYPERMEM-JEV dataset on HuggingFace.

Architectural guidance and runtime infrastructure are provided by DeckerGUI, the design system under which DGUI-HyperMem and the KD ADE Kit are developed. Cloudflare Workers, D1, Vectorize, and Pages form the serverless backbone of the system.

This work is released under the KrackedDevs ecosystem. The KD ADE Kit mobile companion (KD_MATE) is developed and maintained by ctaxnagomi, KD Ambassador for Sarawak, Malaysia.

All product names, logos, and brands are the property of their respective owners. TypeSafe AI, DeckerGUI, KrackedDevs, and CTECX are used for identification purposes only and do not imply endorsement.

References

References

- [1] Cloudflare Inc. D1: Serverless SQLite. <https://developers.cloudflare.com/d1/>, 2025.
- [2] Cloudflare Inc. Cloudflare Workers: Serverless Functions. <https://developers.cloudflare.com/workers/>, 2025.
- [3] TypeSafe AI. JEV: Choice, Noul, and Score Reasoning Layer. <https://typesafe.ai>, 2025.
- [4] DeckerGUI. DGUI-HyperMem: Hybrid Memory MCP Server. <https://github.com/ctaxnagomi/dgui-hypermem>, 2025.
- [5] Google LLC. Flutter: Build apps for any screen. <https://flutter.dev>, 2025.
- [6] George Kour. arxiv-style: LaTeX template for arXiv preprints. <https://github.com/kourgeorge/arxiv-style>, 2014.
- [7] OpenAI. Function calling guide. <https://platform.openai.com/docs/guides/function-calling>, 2025.
- [8] G. V. Cormack, C. L. A. Clarke, S. Büettcher. Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In *Proceedings of the 32nd International ACM SIGIR Conference*, pages 758–759, 2009.