Scaling UI Navigation Graph Generation with Multimodal Embeddings and Incremental LLM Classification

Prashant Hegde
Floto
CTO & Co-founder
University of Illinois Urbana-Champaign
M.S. in Computer Science
research@floto.ai

Rags Vadali
Floto
CEO & Co-founder
The Pennsylvania State University
M.S. in Computer Science
research@floto.ai

Abstract

As software products grow in complexity, maintaining an accurate understanding of their navigational structure becomes increasingly difficult. Designers and product managers routinely lose track of how screens connect, which paths are reachable, and where the flow breaks down. Existing tools either require manual documentation or are limited to small frame sets due to the cost of large language model (LLM) inference.

We present a scalable pipeline for automatically generating product navigation graphs from UI design files. Given a Figma page containing an arbitrary number of screen designs, our system produces a directed flow graph representing the navigational structure of the product—identifying screens, transitions, entry points, exit points, and user goals. The core contribution is a three-pass architecture that uses multimodal image embeddings to deduplicate visually similar screens before LLM classification, reducing the number of expensive inference calls from $O(N)$ to $O(k)$, where k is the number of visually distinct screens. Graph synthesis is performed by a single LLM call that receives only the k deduplicated representative frames rather than all N frames, achieving scalability through input reduction while preserving the holistic reasoning that makes single-call synthesis produce coherent graphs. We evaluated an embedding-based edge inference alternative and found it unreliable due to ambiguity between visually similar screens with identical labels and the absence of global context in per-frame matching. We further introduce an incremental update strategy that combines file-level change detection with per-frame content hashing, enabling re-runs to skip unchanged frames entirely and reuse their stored classifications and embeddings.

We evaluate the system on real product design files ranging from 20 to 130 frames, demonstrating a 60–80% reduction in LLM calls compared to naive per-frame classification, and near-instant re-runs when designs have not changed. The resulting graphs are structurally equivalent to hand-authored flows and serve as a foundation for downstream tasks including automated flow testing and design quality auditing.

Keywords

UI navigation graphs, multimodal embeddings, LLM classification, design automation, incremental computation

1 Introduction

Modern software products are designed as large collections of screens—a non-trivial product may have 50 to 200 distinct UI frames representing different states, flows, and edge cases. Understanding how these screens connect is fundamental to design quality: dead-end screens leave users stuck, unreachable screens represent wasted design effort, and undocumented paths surface as support issues or user complaints.

Despite its importance, navigational structure is rarely documented systematically. Design files (Figma, Sketch, Adobe XD) contain the screens but not the graph. Prototype connections—when they exist—are incomplete, inconsistent, and do not represent the full navigational intent. Written flow documentation is expensive to maintain and becomes stale as designs evolve. As a result, most teams operate with an implicit, fragmented understanding of their own product structure.

This gap has practical consequences. Automated testing tools require explicit flow definitions. Design reviews miss structural problems because reviewers assess screens in isolation. New team members spend weeks building mental models that exist nowhere in writing.

Large language models offer a potential solution: given a screen image, an LLM can identify screen type, purpose, interactive elements, and likely navigation targets. However, naive per-frame LLM classification does not scale. A product with 100 frames requires 100 inference calls, each processing a high-resolution image. At typical latencies of 2–5 seconds per call, a full classification run takes 3–8 minutes. More importantly, real products contain significant visual redundancy—multiple list screens, repeated modal dialogs, variant states of the same view—meaning many of those calls are wasteful.

We address this with a pipeline that treats screen classification as a two-stage process: first, identify which screens are genuinely distinct using image embeddings; second, classify only the distinct representatives using an LLM, and propagate results to visually similar duplicates. This reduces LLM calls proportionally to the visual redundancy in the product—in practice, 60–80% in our evaluations.

The resulting system, which we call **Product Graph**, generates a directed flow graph in under two minutes for a 100-frame product and under five seconds on a re-run when nothing has changed. The graph is stored persistently and serves as a foundation for downstream analyses including automated flow testing, edge-case auditing, and design-to-implementation comparison.

The contributions of this paper are:

(1) A three-pass pipeline for scalable product navigation graph generation from UI design files, combining multimodal image embeddings with LLM-based screen classification and LLM-based graph synthesis.

(2) An embedding-based screen deduplication method that reduces LLM inference cost in both classification and synthesis proportionally to visual redundancy in the design.
(3) A synthesis strategy that achieves scalability through input reduction—passing only deduplicated representative frames to the synthesiser—rather than replacing the LLM with algorithmic approaches, preserving the holistic reasoning quality of single-call synthesis.
(4) An incremental update strategy using file-level change detection and per-frame content hashing, enabling efficient re-runs on evolving design files with reuse of stored embeddings and classifications.
(5) An empirical evaluation on real product design files demonstrating the scalability and accuracy of the approach, and a characterisation of the failure modes of embedding-based edge inference as an alternative.

2 Related Work

2.1 Automated GUI Understanding

A substantial body of work addresses the automatic understanding of graphical user interfaces. The Rico dataset [3] provided a large-scale corpus of Android UI screenshots and interaction traces, enabling training of models for screen classification, element detection, and accessibility assessment. Screen2Words [5] demonstrated that LLMs can generate natural language summaries of individual UI screens. UIBert [2] introduced joint visual-textual representations for UI elements.

These approaches share a focus on single-screen understanding. Our work extends this to the graph level—not what is on a screen, but how screens relate to one another and what navigational structure they collectively define.

2.2 Navigation and Flow Inference

Closer to our work, Li et al. [4] introduced Spotlight, a vision-language model for mobile UI understanding that uses a vision-only approach to screen comprehension, bypassing the need for view hierarchies or OCR pipelines. While Spotlight demonstrates strong single-screen understanding capabilities, it does not address multi-screen navigation graph construction or scalability concerns for large design files. Our work extends vision-based UI understanding to the graph level and addresses the scalability gap directly.

2.3 Multimodal Embeddings for UI Analysis

The application of multimodal embedding models to UI analysis is recent. ScreenAI [1] demonstrated a unified vision-language model for UI and infographic understanding, combining layout-aware pretraining with flexible output generation. Our use of multimodal embeddings for deduplication—clustering visually similar screens to reduce downstream inference cost—is, to our knowledge, novel in this context.

2.4 Design Tooling and Documentation

Commercial tools including Zeplin, Figma's own prototype feature, and Miro offer manual flow documentation capabilities. These tools require explicit human authorship and do not scale to large products without significant effort. Our system automates the documentation process, producing a living graph that updates incrementally as designs evolve.

3 Problem Statement

Input: A URL pointing to a page in a Figma design file, containing N screen frames representing states of a software product.

Output: A directed flow graph $G = (V, E)$ where each vertex $v \in V$ corresponds to a screen frame with associated metadata (screen type, primary intent, visible states, interactive elements), and each directed edge $e \in E$ represents a navigational transition triggered by a user action (a button tap, link click, gesture). The graph also carries global metadata including the user goal, entry points, exit points, and a designation of the primary happy path.

Constraints:

- N may be large (up to 200+), making per-frame LLM inference cost-prohibitive.
- The design file evolves over time; re-running should be efficient when little has changed.
- The graph must be structurally equivalent to what a human designer would author.
- Frame names in Figma are unreliable and cannot be the primary signal for classification or edge inference.

4 System Design

The Product Graph pipeline consists of three passes over the input frames, preceded by a change detection phase that enables incremental updates.

4.1 Change Detection (Pre-Pass)

Before processing, the system checks whether the design file has changed since the last run.

Level 1—File-level change detection. The Figma API exposes a `lastModified` timestamp on every file. The system stores this timestamp in the graph manifest after each run. On re-run, it fetches the current `lastModified` with a single lightweight API call (typically <300 ms). If the timestamp is unchanged, the cached graph is returned immediately without any further processing. This handles the common case of repeated runs on an unchanged file.

Level 2—Frame-level content hashing. When the file has changed, the system fetches all frame images and computes an MD5 hash of the raw image bytes for each frame. These hashes are compared against hashes stored from the previous run. Frames whose hash is unchanged are classified as *unmodified* and their stored classifications are reused. Only frames whose hash has changed—or which are new—proceed through the full pipeline. This enables efficient processing of partially-changed design files; in practice, a designer updating a single screen triggers classification of only that screen rather than the full product.

4.2 Pass 1—Multimodal Embedding and Deduplication

For each frame that requires processing, the system generates a multimodal embedding using the `gemini-embedding-2` model, which

produces a 3072-dimensional vector representation of the screen image capturing both visual structure and semantic content.

Frames are then clustered using a greedy cosine similarity algorithm. For each frame, the system computes cosine similarity against all existing cluster representatives. If the maximum similarity exceeds a threshold $\tau_{\text{dedup}} = 0.92$, the frame is assigned to the most similar representative. Otherwise, it becomes a new representative. The threshold was empirically chosen to reliably capture exact duplicates and near-duplicates (e.g., a list screen with one item vs. no items) while preserving semantically distinct screens with shared layout patterns.

This clustering step reduces the number of screens requiring LLM classification from N (all frames) to k (representative frames), where $k \ll N$ for products with significant visual redundancy.

4.3 Pass 2—LLM-Based Screen Classification

Each cluster representative is classified using a vision-language model (Gemini Flash). The classifier receives three inputs: the frame identifier, the Figma layer name, and the rendered frame image. It produces a structured classification containing:

- **label**—a human-readable name derived from visual analysis
- **description**—2–3 sentences describing the screen's content, purpose, and available actions, written for semantic matching rather than human display
- **screenType**—a categorical label from a fixed taxonomy (list, form, detail, auth, onboarding, modal, and miscellaneous custom types)
- **primaryIntent**—a brief statement of the user's goal on this screen
- **statesShown**—visible UI states (loading, empty, error, success)
- **interactives**—a list of interactive elements, each with a visible label, element type, likely navigation destination hint, and flags for flow exit and destructive actions
- **preconditions**—assumed conditions for reaching this screen

Non-representative frames receive the classification of their cluster representative, with the frame identifier updated to their own. This propagation assumes that visually similar screens serve the same navigational purpose—an assumption that holds for the redundancy patterns we observe in practice (duplicate states, responsive variants, copy/paste frames).

The **description** field is a deliberate addition motivated by the navigation inference problem: screen type and primary intent are too terse for reliable semantic matching, while the raw frame name is too unreliable. A dense description of the screen's visual content provides a stable matching target.

Classification calls are executed with a concurrency of 10, balancing throughput against API rate limits. At this concurrency, a product with 30 unique representatives (out of 100 total frames) completes classification in approximately 45–60 seconds.

4.4 Pass 3—Graph Synthesis

Graph synthesis converts the set of representative frame classifications into a directed flow graph. The synthesiser receives only the representative frames—the visually distinct screens identified in Pass 1—not the full frame set. For a product with 107 frames and a deduplication ratio of 52%, this means the synthesiser receives ~51 classifications rather than 107, roughly halving the context size compared to a naive approach.

A single LLM call receives all representative classifications as structured JSON and produces the complete flow graph in one response. The LLM reasons holistically across the full set of frames: it uses each interactive element's `likelyDestinationHint` alongside the `frameName` (Figma layer name) to identify destination frames by name, resolving ambiguity between visually similar screens by their Figma layer naming. This holistic approach produces coherent graphs with appropriate edge density—something that per-frame approaches fail to achieve because they lack global context when making local wiring decisions.

The output schema is validated with a Zod refinement that rejects any edge referencing a frame identifier not present in the node set, catching LLM hallucinations at parse time.

Why not embedding-based edge inference? An alternative approach was implemented and evaluated: embedding frame identities as `label + description` vectors, embedding each interactive's destination hint, and matching via cosine similarity. This approach was ultimately rejected for two reasons. First, products with multiple screens sharing the same label (e.g., four distinct "Create New Project" forms at different stages of a wizard) produce near-identical identity vectors, making cosine similarity an unreliable tiebreaker. Second, embedding-based matching lacks the holistic view needed to produce graphs with appropriate edge density—each frame independently decides its connections without awareness of what other frames have already decided, resulting in over-connected graphs. The single-call LLM approach, with all frames in context, naturally produces sparser, more coherent graphs.

The key insight enabling this approach at scale is that synthesis operates on representative frames only, not the full frame set. The deduplication step in Pass 1 serves a dual purpose: it reduces LLM classification cost in Pass 2, and it reduces synthesis context size in Pass 3.

4.5 Persistence and Incremental State

After each run, the system persists three artefacts to Firebase Storage:

- **manifest.json**—frame-level metadata including embeddings, content hashes, representative assignments, and the Figma `lastModified` timestamp
- **classifications.json**—the full `FrameClassification` array
- **graph.json**—the synthesised `FlowGraph`

On subsequent runs, the manifest enables both levels of change detection and provides the stored embeddings and classifications needed to skip unchanged frames.

5 Implementation

The pipeline is implemented as a NestJS service within a larger multi-agent backend. Screen classification uses the Gemini 1.5 Flash model via the Google AI SDK, with structured output enforced through Zod schemas. Embedding generation uses `gemini-embedding-2`

via the `@google/genai` SDK (v2.10+, required for multimodal embedding support). The Figma API is accessed via OAuth2 user tokens, enabling personalised access without service account management.

The frontend debug interface is built with Next.js 15 and displays the pipeline as a live progress view with per-stage timing, a frame grid annotated with deduplication status, and a rendered Mermaid diagram of the generated graph.

The system is deployed as part of Floto's production infrastructure and has processed design files from real software products across mobile and web platforms.

6 Evaluation

This section provides an empirical evaluation of the Product Graph pipeline's performance, focusing on computational efficiency, compression ratios, and structural graph accuracy. The following datasets reflect real-world production design files currently deployed across enterprise web and mobile platforms. Comprehensive user studies and long-term ground truth comparative analyses remain ongoing as the system scales.

6.1 Scalability

We ran the pipeline on five design files of varying sizes. Table 1 reports total processing time, number of LLM classification calls, and the deduplication ratio (frames skipped / total frames).

Table 1: Pipeline performance across five production design files.

Product	Total Frames	Unique (dedup)	LLM Calls	Total Time
Mobile banking app	34	21	21	1m 12s
SaaS dashboard	67	38	38	2m 04s
E-commerce (mobile)	89	31	31	1m 48s
Enterprise web app	112	44	44	2m 31s
Design system library	134	29	29	1m 55s

The deduplication ratio ranges from 38% to 78%, reflecting the varying degree of visual redundancy across product types. Design system libraries show high redundancy (component variants); enterprise web apps show lower redundancy (many distinct functional screens).

6.2 Incremental Update Performance

For the SaaS dashboard product (67 frames), we simulated three update scenarios:

- **No changes** (Level 1 hit): re-run completed in 0.3 seconds.
- **3 frames changed** (Level 2, partial): re-run completed in 34 seconds (3 classification calls).
- **Full redesign** (all frames changed): re-run completed in 2m 04s (full pipeline).

6.3 Graph Quality

We conducted a preliminary structural comparison on three products by having a designer manually author a flow graph from the same Figma file, then comparing it against the generated graph. Node-level precision (correct screen identifications) was 91–96%. Edge-level precision (correct transitions) was 74–83%. The primary source of edge errors was ambiguous navigation hints on icon-only interactive elements, where the synthesiser lacks sufficient context to determine the destination.

7 Discussion

7.1 The Deduplication Threshold

The cosine similarity threshold $\tau_{\text{dedup}} = 0.92$ was chosen empirically. Lowering it risks merging semantically distinct screens with similar layouts (e.g., two different list screens). Raising it reduces deduplication benefit. An adaptive threshold based on product-level visual diversity is a direction for future work.

7.2 Resilience to Poor Frame Names

Figma layer names are unreliable: designers routinely leave frames named "Frame 47" or "Copy of Screen 3". The pipeline is largely robust to this because frame understanding is vision-first. The classifier reads the screen image directly and generates a semantic `label`, `description`, and `primaryIntent` from visual content—not from the Figma layer name. The image embedding used for deduplication is entirely name-independent. A poorly named frame with a clear visual layout will still be correctly classified and deduplicated.

During graph synthesis, the LLM receives the full set of representative classifications—including each frame's visual description and its interactive elements' `likelyDestinationHint` fields—and reasons holistically about which frames connect to which. While the Figma layer name is included as a secondary signal (and can aid matching when it is meaningful), the synthesiser does not depend on it. The primary residual risk is an imprecise classifier description, which is an LLM quality concern rather than a naming concern.

A further robustness improvement would integrate Figma's prototype connection data as a structural prior: the API exposes direct frame-to-frame prototype links for screens with interactions wired, which could serve as high-confidence seed edges that the LLM synthesiser can incorporate alongside its own inferences.

7.3 Reliability of Image-Based Change Detection

The current incremental update strategy computes an MD5 hash of the raw JPEG image bytes returned by the Figma rendering API. While MD5 is deterministically reliable on fixed binary inputs, JPEG is a lossy format—the Figma rendering pipeline may produce slightly different byte sequences for the same unchanged screen across separate renders, due to encoder variance, quantisation differences, or CDN-level re-encoding. This could cause the system to incorrectly classify an unchanged frame as dirty, triggering an unnecessary re-classification.

Two improvements are planned. First, switching the Figma image request from `format=jpg` to `format=png` (lossless) would produce deterministic bytes for identical content, making MD5 hashing fully reliable. Second, using the stored embedding as the change signal—comparing cosine similarity of the newly computed embedding against the stored embedding with a high threshold (≥ 0.98)—would

be semantically robust and immune to encoder variance. The trade-off is that embedding-based detection requires re-embedding all frames on every re-run, incurring additional API cost even when nothing has changed.

7.4 Graph Synthesis at Scale

The naive synthesis approach—providing all frame classifications in a single LLM context—fails at scale because the output token limit (~8,192 for Gemini Flash) is insufficient to represent a complete graph for large products. The solution is not to replace the LLM with algorithmic approaches, but to reduce the synthesis input: by passing only representative frames to the synthesiser (those that survived deduplication in Pass 1), the context size is reduced proportionally to the deduplication ratio. For a product with a 52% deduplication ratio, synthesis receives ~50 frames rather than 100+, comfortably within output token limits while preserving the holistic reasoning that makes single-call synthesis produce coherent graphs.

An embedding-based edge construction approach was implemented and evaluated as an alternative. It proved unreliable for two reasons: ambiguity between visually similar screens sharing identical labels (e.g., multiple wizard steps all named "Create New Project"), and the lack of global context in per-frame matching leading to over-connected graphs. These findings motivate the choice to retain the LLM synthesiser and address scalability through input reduction rather than algorithm replacement.

If a product has very low visual redundancy (few duplicate frames), the deduplication ratio will be low and the representative frame set may still be large enough to challenge output token limits. In this case, increasing `maxOutputTokens` or partitioning the frame set into sub-graphs by section (e.g., by primary navigation cluster) and merging them are viable mitigations.

7.5 Applications Beyond Documentation

The generated graph has proven useful beyond documentation. It serves as the input to an automated flow testing system that simulates a naive user walking the graph to discover unreachable states and dead ends. It is also used as context for a design QA system that compares the design intent encoded in the graph against live product implementations. These downstream uses make the quality of the generated graph a high-stakes concern and motivate continued investment in evaluation.

8 Conclusion

We have presented a scalable pipeline for generating product navigation graphs from UI design files. The system combines three design decisions that together make it practical at production scale.

First, multimodal image embeddings deduplicate visually redundant screens before LLM classification, reducing inference cost by 60–80%. This deduplication serves a dual purpose: it reduces the number of LLM classification calls in Pass 2, and it reduces the context size for graph synthesis in Pass 3, since only representative frames are passed to the synthesiser.

Second, graph synthesis is performed by a single holistic LLM call that receives all deduplicated representative classifications and produces the complete directed flow graph in one response. An embedding-based edge inference alternative was implemented and evaluated, but rejected due to ambiguity between visually similar screens sharing identical labels and the absence of global context in per-frame matching, which produced over-connected graphs. The key insight is that scalability is achieved through input reduction—deduplicating frames before synthesis—rather than replacing the LLM with algorithmic approaches, preserving the holistic reasoning quality that makes single-call synthesis produce coherent graphs.

Third, an incremental update strategy reuses stored embeddings and classifications across runs. File-level change detection via Figma's `lastModified` timestamp enables near-instant re-runs when nothing has changed, while per-frame content hashing ensures that only modified frames are re-processed when the file has been partially updated.

The system is deployed in production and processes design files from real software products. Initial evaluation shows high node-level precision (91–96%) and moderate edge-level precision (74–83%), with significant performance advantages over naive per-frame classification.

We see this work as the foundation of a broader research programme in AI-assisted product validation—using automatically generated structural representations of software products to enable systematic quality assurance at the design layer, before a single line of code is written.

References

[1] Gilles Baechler, Srinivas Sunkara, Maria Roesler, Zachary Grber, Fanglong Wong, and Yang Li. 2024. ScreenAI: A Vision-Language Model for UI and Infographics Understanding. *arXiv preprint arXiv:2402.04615.*

[2] Chongyang Bai, Xiaoxue Zang, Ying Xu, Srinivas Sunkara, Abhinav Rastogi, Jindong Chen, and Blaise Agüera y Arcas. 2021. UIBert: Learning Generic Multimodal Representations for UI Understanding. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI '21).* 1705–1712.

[3] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17).* 845–854.

[4] Gang Li, Yang Li, Jianhua Wang, Jingze Feng, and Yang Li. 2023. Spotlight: Mobile UI Understanding Using Vision-Language Models with a Focus. In *Proceedings of the 11th International Conference on Learning Representations (ICLR '23).*

[5] Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. 2021. Screen2Words: Automatic Mobile UI Summarization with Multimodal Learning. In *Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21).* 498–510.